Uniqueness and Lazy Graph Copying

Copyright for the Unique

Marco Kesseler
Faculty of Mathematics and Computer Science
University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands

September, 1994

Abstract

The uniqueness type system and lazy graph copying are important techniques to efficiently implement functional languages. Unfortunately combination of both in one system may lead to conflicts. Until recently, parallel Concurrent Clean programs could not take advantage of the uniqueness type system, because the lazy graph copying method that Concurrent Clean employed was able to invalidate derived uniqueness information. This paper will address this problem and present a solution that is based on a new copying method with different semantics, called lazy normal form copying.

1. Introduction

The uniqueness type system and lazy graph copying are important techniques to efficiently implement functional languages [1, 2, 3]. On the one hand, uniqueness information allows the compiler to employ destructive updates in certain cases. This forms the basis of efficient array implementations and the Concurrent Clean IO system. On the other hand graph copying is needed for implementations on parallel machines with distributed memory.

Graph copying is an extension of the standard graph rewriting semantics. This means that one has to make sure that the semantics of both do not interfere. Unfortunately, graph copying may conflict with the uniqueness type system. This becomes apparent if one considers that duplication of data may invalidate uniqueness properties. How can some unique data structure remain unique after copying? If this cannot be achieved one would have serious problems to efficiently incorporate arrays and the Clean IO system in a parallel environment.

This paper will identify possible problems and propose a solution. It will show that difficulties mainly arise in situations where copying gets postponed (deferred) and that graph copying in itself is not dangerous with respect to uniqueness. The graph copying method that has been employed in Concurrent Clean so far turns out to be incompatible with the uniqueness type system. We will present a new copying method with different semantics - called lazy normal form copying - that solves these problems by maintaining all derived uniqueness properties.

In addition, we will see that lazy normal form copying enables the programmer to identify more clearly what gets copied. In short, lazy normal form copying does not copy work unless stated explicitly. By default only normal forms are copied. This means that the type of a function indicates the structure of the data that will be copied. We will see later what this implies for writing parallel Concurrent Clean programs.

We have structured this paper as follows. First, we will take a closer look at uniqueness properties in Concurrent Clean and clarify the dangers of lazy graph copying. Next, a number of possible solutions will be discussed, leading to the one we have adopted. Finally, some examples will show how this affects programs written in Concurrent Clean.

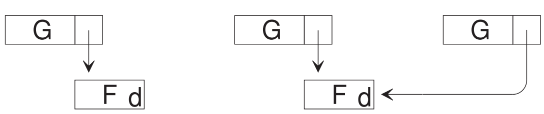
2. Lazy Copying affects Uniqueness Properties

The Concurrent Clean uniqueness type system [4] is based on the observation that arguments can be locally unique to a function, which means that only this function has direct access to the argument. The function itself may not be uniquely used, so uniqueness is a local property. If a function has unique arguments, it may destructively update these arguments with (part of) the function result. This is safe, as no other function is be able to access any unique argument without evaluating the parent function first: one cannot traverse a closure. Constructors, on the other hand, can be traversed, but for these the uniqueness type system requires that the constructor itself must be unique if one of its arguments is unique. Thus uniqueness of arguments with respect to the enclosing function is guaranteed, which implies that destructive updates of unique data do not hurt referential transparency.

Basically two types of unique data structures exist: ordinary unique data and essentially unique data. Both can be used to implement destructive updates. The only difference between the two is that essentially unique data - as opposed to ordinary unique data - cannot be coerced to a non-unique type. The essentially unique type is needed for curried functions that have unique arguments [4]. Essentially unique data cannot become shared, and copying essentially unique data is more problematic than copying ordinary unique data.

Lazy graph copying is an extension of standard graph rewriting semantics. If one combines it with uniqueness types, one has to make sure that copying either does not affect the derived uniqueness information in an unsafe way, or that it is able to adjust uniqueness properties on the fly. Not all changes are dangerous. A non-unique object becoming unique is no problem, except perhaps for not exploiting this information (It would be interesting to investigate the possibilities for optimisations when nodes become unique after copying, but we will not consider this here). In contrast, a unique object becoming non-unique will result in serious problems if this change is not taken into account by functions that refer to this object. Objects that are actually not unique might then be updated in place, which destroys referential transparency.

The lazy graph copying method that has been used in Concurrent Clean until recently does not maintain uniqueness properties. Graph copying in itself does not impose serious problems: if one copies a graph as a whole, the copy will be exactly the same graph as the original and uniqueness properties of individual nodes will be the same as for the original. If an argument is unique with respect to some function, the copy of the argument will be unique with respect to the copy of the function. Problems arise when introducing laziness to copying. Lazy copying will not always result in physical copy. Instead, it will maintain sharing as much as possible. An extreme example is copying a (sub)graph to the same processor it resides on: the resulting ‘copy’ will be a shared graph and not a duplicated one. The extra references this introduces may invalidate the uniqueness information that has been derived by the Concurrent Clean compiler.



Picture 1: Lazy copying changes derived uniqueness properties. F is a function that delivers a unique result. G takes this unique result for its argument. If we have the following expression (h, {P} h), with h defined as G {I} F, the application of F will be deferred - as it is being reduced by a separate process - and copying h will stop at F. The leftmost figure shows the graph h before copying, the rightmost figure shows the original h and its copy. Clearly F is no longer unique with respect to G.

This leaves us with the question in which cases problems arise. If one stops copying at a unique node, a new reference to this node will be created and uniqueness may be destroyed. In Concurrent Clean a special node attribute - the defer attribute - indicates whether copying should (temporarily) stop or not. Thus, only deferred unique nodes are problematic.

On the other hand, not all deferred unique nodes will introduce conflicts, but only those contained within a closure that gets copied. If no closure around a deferred unique graph gets copied - for instance when the copied graph is in normal form - no problems occur due to the upward uniqueness propagation of constructors: all copied nodes around the unique part will be unique as well. This means the original graph will become garbage after copying has succeeded, because the copy function delivers the copy and discards the original. Therefore the newly created pointer to the unique deferred substructure will become the only one left after copying and uniqueness is not violated[1]. This shows that conflicts are closely related to the exclusive ability of the graph copier to traverse closures. These observations will turn out to be important for the solution we have adopted.

2.1. Possible Solutions

Simply banning laziness from the graph copier to avoid problems will not be wise: it is an important technique to share results as opposed to recomputation. For instance, programs that employ processes to consume the results of other processes rely on it: instead of copying work back, lazy copying stops the consumer as soon as it hits a node the producer is working on. In addition, lazy copying is required for structures of the Concurrent Clean IO system that are part of the operating system that it runs on. For these structures special rules apply as they usually refer to physical objects such as disk drives, screens, and windows. For sequential programs it is sufficient to assign unique types to these objects if they should be updated in place (for example when writing to a disk). For parallel programs some extra rules are needed. Although sharing is possible, one cannot copy objects such as disk drives. This means that these objects are not essentially unique (they can be shared), but they cannot be copied as ordinary unique objects can: they remain deferred. In this way one can duplicate references to such objects, but the objects themselves should become non-unique and stay at their physical location.

Keeping laziness, the current copy function is able to change the actual uniqueness properties of graphs during runtime when certain objects are encountered without adjusting the derived uniqueness information. Three possible remedies come to mind. First of all one could implement runtime coercions to make functions aware of the changes that have occurred. Secondly, one could avoid the creation of objects that may introduce uniqueness conflicts during copying. And finally, one could change the semantics of the copy function so that it cannot change uniqueness properties at runtime.

The first solution would imply that functions invoke evaluation code that depends on the actual runtime uniqueness properties. Evaluation code addresses of closures would have to be

[1] A similar situation exists if a unique argument is surrounded by closures that become garbage after copying, but this is not enforced by the uniqueness of the argument, so it will not always hold.

modified by the graph copier if it detects that an argument is no longer unique. Not only is this needed for the nodes in the copy but for nodes in the original graph as well. Such a solution is intolerable: copying becomes (even) more complex requiring expensive runtime checks for uniqueness properties and it gives rise to unclear runtime behaviour. This surfaces most clearly when essentially unique objects are involved. These cannot be coerced to a non-unique type and a runtime error would be the result.

This leaves us with two directions that may lead to a useful solution. Before considering how a copying method that preserves uniqueness properties is obtainable, we will focus on the possibility of avoiding creation of deferred unique objects, that is, objects that may introduce uniqueness conflicts.

3. Avoiding Deferred Unique Objects

Only deferred unique nodes may cause copying problems. Avoiding creation of these objects will solve the copying problems. This is discouraging on beforehand: in some situations it is worthwhile to have these objects and there is no reason that they will actually result in conflicts, so avoiding them regardless will often be harmful and of no use at all. Keeping this in mind, we will examine the feasibility of this method in more detail below.

Suppose one wants to avoid creation of deferred unique objects, either during runtime, but preferably at compile time. How can this be achieved? The compiler is able to reject the use of unique types at places it needs to defer nodes to avoid duplication of work. At runtime on the other hand, one should take care of the cases the compiler cannot detect, which means one has to observe that unique nodes do not become deferred. We will have a look at both cases below.

3.1. Reject uniqueness for deferred objects

The compiler can detect creation of deferred objects in a number of cases. Rejecting unique types in these cases would avoid conflicts during copying.

- During normal graph reduction nodes get reserved (and thus deferred).
- During creation of a cycle the empty nodes are deferred.
- The {I} and {P} annotations for parallelism will defer the root of the annotated graph.
- Structures of the IO system that cannot be copied are deferred.

The first case seems to indicate that the use of unique types becomes virtually impossible. This turns out not to be the case if one takes a closer look at the order in which nodes get evaluated. When a reducer reduces (and reserves) a unique argument during evaluation of some function f it will have reserved the surrounding closure f as well. Any access to the unique argument must pass f. This holds for the graph copier as well, and as the parent closure is reserved copying will stop there automatically and not at the argument, so deferring the unique argument is safe. Clearly the order of evaluation is crucial here, considering the problems that occur when annotations for parallelism are used. As a result the compiler only has to avoid unique types in the remaining three cases.

This approach has serious drawbacks. Not only does one risk a notable performance penalty if processes are not allowed to deliver unique results, but the Concurrent Clean IO system would become useless if it cannot use unique structures in certain cases. Note furthermore that the {I} annotation in itself has no influence whatsoever on the uniqueness properties of graphs: no two processes running interleaved at the same processor are able to access unique arguments of the same closure, because closures are locked during reduction. This is in accordance with the view that processes do not introduce new data dependencies, and only provide safe eager evaluation (safe with respect to program termination). In effect, the introduction of processes can be seen completely separate from the (unwanted) effects of graph copying.

3.2. Avoid deferring unique objects at runtime

Sometimes it is very useful to defer additional nodes temporarily to stop copying a graph at some point. For instance, this may be advisable when the copy of a graph is about to become very large. After copying a certain amount of nodes, one would like to stop and copy the rest only when needed. Likewise one also would like to stop copying when a large flat data structure - such as a strict array - is hit. This will automatically create a separate message for such a structure, which can be more efficient than packing it in a message along with the other nodes of the copied graph [5].

Clearly, this cannot be done for unique objects, as they would become deferred. One has to detect and avoid this during runtime, because the compiler cannot detect such introductions of temporarily deferred nodes. Regardless whether this is feasible in an efficient way or not, it would destroy the possibility to use these optimisations for unique objects. Even if absence of optimisations for unique objects does not cause considerable problems in practice, one still faces the problem of unclear runtime copying semantics[2].

4. Uniqueness Preserving Copying

From the above, it has become clear that avoiding the creation of dangerous objects has serious drawbacks, mainly because these objects are very useful to have for various reasons, while they may not become problematic at all. Instead, we will design a graph copier that does not alter uniqueness information. Laziness will not be discarded. In contrast, copying will become even lazier as it will stop before it hits nodes that introduce conflicts. This section will show how this can be achieved. Our solution is based on the observation that no conflicts arise as long as no closure is traversed during copying. The new copier will decide to stop copying at closures, depending on their use. As (the absence of) copying closures plays an important role, we will concentrate on copying closures first.

4.1. Copying Closures

When is it safe to copy a closure? It has been argued earlier that the ability to traverse closures during copying gives rise to conflicts. As long as no closure is traversed no conflicts occur. Not all closures are dangerous however, only those with unique arguments. Only by traversing such a closure one can add references to its unique arguments. In addition, closures

---

[2] A related problem occurs if one stops copying at a closure. What should happen later if this closure is requested? Should it be copied and reduced by the requesting process, or should it be reduced locally so that the result can be returned. The latter would ensure the closure will always be evaluated at the same place, but this would not have happened if copying had proceeded.

with unique arguments are not always a threat. Firstly, the total amount of pointers to any unique deferred argument will not change if one can assure that the original closure becomes garbage after copying. Secondly, if it is known that a unique argument will be copied entirely, copying will not be deferred and the amount of pointers remains the same for both copied and original unique argument.

Unfortunately, the compiler does not know in general if an argument of some closure can be copied entirely. Amongst others, this would involve knowing whether the argument is being reduced or at which processor it is. In addition the compiler usually cannot determine if a closure will become garbage after copying. On the one hand, closures do not propagate uniqueness information upwards in the way that constructors do (except if they deliver essentially unique results). On the other hand, a closure that delivers a totally unique structure may very well contain a non-unique subgraph that is reachable from ‘outside’ the graph itself (see figure 2). As a result, one cannot tell at compile time exactly which closures are safe to copy and which not. Only an approximation would be possible, presumably complemented by runtime checks.

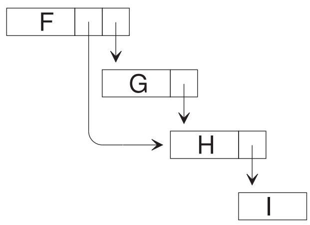


Figure 2: Closures obstruct propagation of uniqueness information. Assume G builds a unique result, but it does not have a unique argument. In contrast, H has a unique argument, but it does not deliver a unique result. The expression F c (G c) with c defined as (H I) results in a graph as depicted. The graph rooted by G is unique, but the subgraph rooted by H is not. In short, uniqueness of a graph does not imply that subgraphs are unique, and uniqueness of a subgraph does not mean that surrounding nodes are unique.

Having copying decisions that depend on complex compile time analysis and even runtime checks will make reasoning about copying behaviour an impossible task. This is especially problematic for closures as these represent work. Any copying decision that involves closures will either move work, copy it, or leave it at its place. Unclear copying semantics will make it hard for a programmer to figure out what will happen where. The next section will show the approach we have chosen to solve this.

4.2. Lazy Normal Form Copying

To avoid the problems listed above, we have taken a radical approach that does not copy closures (work) at all, unless explicitly requested. Default copying will stop at closures. If a programmer wishes to copy closures she (or he) has to indicate them explicitly. This could be done with annotations, by means of special type denotations, or otherwise. We propose a new {P} annotation that will copy only one closure: the annotated root closure. Copying may proceed at the arguments of this closure, but only as long as they are in root normal form. An example below will clarify this.

Using this new {P} annotation, the compiler should check if the annotated root closure can be copied safely. Criteria for copying closures have been defined above. We will not try to exploit all possibilities, but instead we have chosen to keep the copying decision safe and simple (for ease of reasoning about programs). An annotated closure may only be copied if:

- it has no unique arguments
- it has unique arguments and it becomes garbage after copying.

The first check is trivial. The last check is often easy for annotated root nodes, because most of the time the annotated root will be a locally created function. If it is not locally created, it will have been passed as a parameter and the compiler should inspect the uniqueness properties of the passed closure with respect to the current function. If the annotated closure turns out to be non-unique, copying must be rejected. One could consider this an invalid coercion of types.

To avoid creation of empty copies, which would cause deadlocks, the copier will start up new processes to drive reduction: if copying stops at the root of a graph a reducer will be started on it and copying proceeds after reduction to root normal form. In this way remote processes are able to drive a local computation in a lazy manner and vice versa.

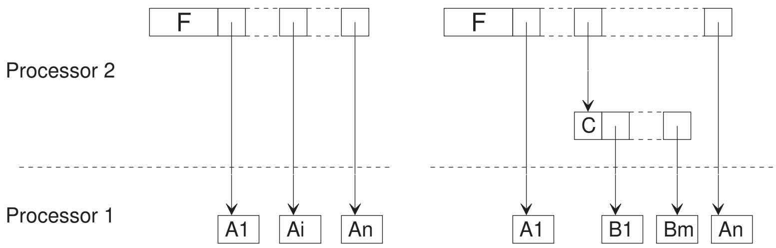


Figure 3: A1, A2 ... An are all graphs that are not in root normal form. The leftmost picture shows the distribution of graphs after evaluating the expression ({P} F A1 A2 ... An) at processor 1. F is the only closure that has been copied to processor 2. If it needs argument Ai, it will send a request for it. The graph copier will then start a new process on Ai and return the result as soon as it has been computed. Suppose this is a constructor C with arguments B1 B2 ... Bm that are not in root normal form. Only the constructor will then be copied as shown in the rightmost picture. If some Bj is needed later, the same will apply as for Ai, etc. Note that if Ai was already in root normal form before copying F, the rightmost graph would have been obtained directly.

The advantages are obvious. In addition to avoiding uniqueness conflicts, the runtime behaviour becomes more clear, as implicit - and conditional - copying of work is avoided. Work sticks at its location. Without special measures one can be sure that only normal forms are copied. If one uses explicit closure-copying to evaluate a function F at another processor to deliver some structure in parallel, one can be sure that the structure will be returned completely evaluated and not some function that computes (parts of) it. The same holds for the arguments passed to F: these will be evaluated locally so that the remotely evaluating F will get evaluated arguments and not extra work. One does not risk to copy more work than intended. In effect the type of a graph indicates what will be copied eventually. With the old lazy copier a similar copying behaviour could only be obtained by using extra annotations to drive computation explicitly, but this did not result in distributed lazy evaluation.

There are some drawbacks as well. Copying of work becomes more difficult. Suppose one has an argument a and one wishes apply F·G at another processor. The {P} annotation only copies the root closure, so that {P} F (G a) means that only F is computed at the other processor, while G is evaluated locally - either on beforehand if F is strict in its argument, or later when needed. Some extra work is needed to specify the demanded behaviour. One may define a new function H: H a -> F (G a). {P} H a would now perform as intended.

The introduction of such functions seems to be problematic in certain cases. Take the following example, where H could be some skeleton for parallelism. Annotations are bold to distinguish them from ordinary language constructs. The argument function g will not be evaluated at the same processor as F. Suppose it should. How does one accomplish this? Perhaps an extra annotation would be needed. We will take a look at this next and show that this is not necessary, as the use of curried functions provides a powerful means to copy work explicitly.

```
:: H (x->y) x   -> y;
   H g a        -> {P} F (g a);
```

4.3. Currying and Copying

The use of curried functions forms an interesting area with respect to graph copying. The type of a curried function is denoted as (A_1 A_2 ... A_n -> R). If the type of a graph should indicate what gets copied, one could copy a function that takes arguments of type A_1, A_2, ..., A_n, and delivers a result of type R. In contrast, one could also decide not to copy curried functions as these represent work just as well as ordinary functions. We have chosen for the former solution as it provides a clear and powerful way to copy work safely.

The uniqueness type system treats curried functions almost like constructors. If a curried function has unique arguments, it gets an essentially unique type. This ensures that uniqueness is propagated upwards. As a consequence, the use of curried functions does not introduce copying problems. Furthermore, in Concurrent Clean the low level representation of curried functions is very similar to the representation of constructors. This allows the new copying function to automatically handle curried functions the right way, that is, to copy them.

Using currying to pass work to other processors, the previous example becomes as follows. Here we can see how the curried function type clearly indicates that work will be copied.

```
:: H (x->y) x          -> y;
   H g a               -> {P} ApplyFG g a;

:: ApplyFG (x->y) x   -> y;
   ApplyFG g a         -> F (g a)
```

This way of passing work to other processors has the inconvenience that one cannot use it directly to pass closures without any arguments. It is possible to work around this problem by using an extra function with a dummy argument, but this would not be an elegant solution. It remains to be seen how this can be solved in a better way.

5. Using Lazy Normal Form Copying

Due to different copying semantics the way parallel programs are written in Concurrent Clean will change. The previous example already makes this clear. This section will show how some other programs are affected. Perhaps superfluous to mention: using another graph copier does not affect the final outcome of programs. Changes are only needed to influence efficiency.

5.1. A Simple Divide and Conquer Program

Take a divide and conquer program to count the occurrence of some element in a tree. Using the old lazy copying semantics this could have been defined as follows.

```
Count elem NilTree
   -> 0;
Count elem (Node elem' left right)
   -> ++ countrest,      IF = elem elem'
   -> countrest,
      countrest:  + ({P}Count left) ({P}Count right);
```

Here we can see the danger of implicitly passing extra work to each processor that evaluates a count function in parallel. The old copying function copies closures that are contained in the subgraphs, for instance elements that have not yet been computed. These may represent a substantial amount of work. To avoid this problem one had to evaluate the tree explicitly - using annotations - before passing it to the counting function. The new graph copier does not have this problem. One can determine locally what work will be performed by another processor.

But suppose one wants to evaluate the elements of the tree at the processor that executes the count function. How does one accomplish this? Again, we will not invent extra annotations, but resort to the use of currying. If the elements of the tree must be computed at another processor, the type of the tree must be changed. One could use the following example.

```
:: Tree x ->   Node (Tree x -> x) (Tree x) (Tree x);
               NilTree;
```

The curried function indicates the function that should be applied to the tree itself to get the element of its root node. The type of this curried function is rather arbitrary. It could have been defined having a different type of argument or a different number of arguments. Unfortunately, the number of arguments cannot be zero. The second rule of the count function would now become as indicated below. Computing the element has become explicit. The definition of the curried function precisely states what will happen if it is evaluated at another processor. Note that the arguments of the curried function will not necessarily be evaluated at the same processor as the curried function itself. For these the same rules apply as for annotated functions.

```
Count elem n:(Node elemfunction left right)
   -> ++ countrest,      IF = elem (elemfunction n)
```

```
      -> countrest,
         countrest:  + ({P}Count left) ({P}Count right);
```

In some cases it will be more efficient to perform this function on a graph that has been distributed over a network already. The count function should then be started at the location of its argument. In this case there would be no difference between the old and the new copy function as both would merely copy the count function. At this moment there is no easy way to express this in Concurrent Clean. This will possibly be addressed by a special apply function.

5.2. Nfib

This is a notorious benchmark. It is approximately defined as follows.

```
Nfib 0   -> 1;
Nfib 1   -> 1;
Nfib n   -> ++ (+ (Nfib (-- n)) ({P}Nfib (- n 2)));
```

The original copying function would cause the whole expression (Nfib(n-2)) to be sent to another processor - using a single message -, which is very efficient. In contrast, the new copier will keep (n-2) local and it will only be evaluated (and copied) after the nfib function requests its remote argument. This clearly is less efficient. To obtain the original behaviour one could use an intermediate function.

```
Nfib n   -> ++ (+ (Nfib (-- n)) ({P}Nfib' n));
Nfib' n  -> Nfib (- n 2);
```

Another efficient solution would be the use of a strictness annotation as shown below. This will force (n-2) to be evaluated before creating a new process. Currently the Concurrent Clean system does not derive strictness information for expressions that have annotations for parallelism. Note that the location where (n-2) is evaluated does not change by placing a strictness annotation, which would have happened if the old copying semantics had been used.

```
Nfib n   -> ++ (+ (Nfib (-- n)) ({P}Nfib !(- n 2)));
```

5.3. The Sieve of Erathostenes

The advantages of lazy normal form copying surface clearly when using some form of (lazy) stream processing, of which the sieve of Erathostenes is a well-known example. The old copying semantics required the programmer to place annotations for two reasons. First of all to drive computation, and secondly to defer copying at certain closures. This can be seen in the example below, where an {I} annotation is needed at a filter function to keep it at the current processor when a new parallel sieve is started.

```
Sieve [p | s]  -> [p | {P} Sieve {I} (Filter s p)];
Sieve []       -> [];

Filter [x | xs] p
```

```
    -> Filter xs p,         IF (= (% x p) 0)
    -> [x | {I} Filter xs p];
Filter [] p -> [];
```

Due to the new copying semantics the first {I} annotation is not needed anymore. The filter function will remain at the correct location. A new process will automatically be started on the filter function if its result is needed by the next sieve. If one does not care about speed of execution and just wants to evaluate the sieve in a distributed lazy manner, no {I} annotations are needed in the filter function either.

Using a special list buffering function to control communication and synchronisation [5] the definition of the sieve would become as follows. The filter function does not require any annotations in this case, because now the buffer function drives evaluation. Tests of preliminary versions of such buffers turned out to give notable speed-ups for the sieve program: up to 8.9 on 16 processors and 15.7 on 32 processors.

```
Sieve [p | s]  -> [p | {P} BuffSieve (Filter s p)]
Sieve []       -> [];

BuffSieve list -> Sieve (Buffer buffersize list),
   buffersize: 40;

Filter [x | xs] p
-> Filter xs p,      IF (= (% x p) 0)
-> [x | Filter xs p];
Filter [] p -> [];
```

6. Conclusions

Lazy graph copying is an extension of the standard graph rewriting semantics. We have shown that it may conflict with the uniqueness type system. This depends on the exact semantics of the copying function. Until recently, Concurrent Clean employed a graph copying method that was incompatible with the uniqueness type system. It could invalidate uniqueness information during runtime. This paper has addressed this problem and presented a new copying method with different semantics that does not alter uniqueness properties. Additionally, this new copying method does not copy work implicitly. This allows programmers to reason more easily about the exact behaviour of parallel programs.

7. References

[1] Plasmeijer M.J., Eekelen M.C.J.D. van (1993). *Functional Programming and Parallel Graph Rewriting.* Addison Wesley, 1993.

[2] Nöcker E.G.J.M.H., Smetsers J.E.W., Eekelen M.C.J.D. van, Plasmeijer M.J., (1991). 'Concurrent Clean', In *proceedings of Parallel Architectures and Languages Europe (PARLE'91).* Springer LNCS 505, Vol. II, page 202-219.

[3] Eekelen M.C.J.D. van, Plasmeijer M.J., Smetsers J.E.W., (1990). 'Parallel Graph Rewriting on Loosely Coupled Machine Architectures' In *proceedings of the workshop on CTRS'90.* Montreal Canada.

[4] S. Smetsers, E. Barendsen, M. van Eekelen, R. Plasmeijer (1993). Guaranteeing Safe Destructive Updates through a Type System with Uniqueness Information for Graphs. In *Graph Transformations in Computer Science.* Dagstuhl Castle, Germany, Springer LNCS 776, pages 358-379.

[5] M. Kesseler (1994). Reducing Graph Copying Costs - Time to Wrap it up. In *Proceedings of the First International Symposium on Parallel Symbolic Computation (PASCO '94),* Linz, Austria, World Scientific International Co. To Appear.