Thesis Proposal:

Dataflow Analysis for Recursive Structures

Tim Lewis

June 2, 1997

1 Introduction

Imperative languages allow unrestricted graph structures using pointers which makes their static analysis difficult and often imprecise. An approach is proposed whereby data structures are described based on a *path name*. This name uniquely determines a node of the structure and must not depend on run time information. The aim is to develop a method for precise static analysis for a class of structures with this node-naming property.

The data structures considered are graphs with the data elements stored at the nodes, and each node has a fixed set of outgoing directions D. The nodes can be indexed by the language L over D, each word being a path name from a distinguished root node denoted by ϵ. There is also an equivalence relation, $\equiv$, over the words of L, which describes when two different paths actually reach the same data node. This can be expressed as a finite set of relations $r_i \in (L \times L)$, *i.e.* the relation $r_i = (x_i, y_i)$ implies that $u.x_i.v \equiv u.y_i.v$ for all $u, v \in L$. The set of names v such that $u \equiv v$ is the set of *aliases*. It should be noted that such a system would not necessarily have a decidable word problem. An example of such a set of relations that does is the properties: *inverses* $r_i = (ab, \epsilon)$, and *commuting pairs* $r_i = (ab, ba)$, where $a, b \in D$. We can define a normalising function n on such path names, so that all paths can be mapped to a unique equivalent one, *i.e.* it has the property that $x \equiv y \iff n(x) = n(y)$. In order to handle more general structures we would use the normalising function as the definition, rather than the relations. A Markov algorithm [2] is proposed as a formal way of doing this.

The programs under consideration are recursive functions with a path name as the parameter w, as shown in the example in figure 1. Statements in the body of the function can read and write to elements of the structure obtained by prefixing the parameter path name by a word from L. Additionally, we can call the function recursively with the new parameter $w.x$, for any direction x. During the analysis each recursive call in the function body is assigned a label from an alphabet $\mathcal{C}$. Also, each read and write statement is given a label from $\mathcal{I}$; we will often only consider one read and one write denoted by R and W. A *control word* in $\mathcal{C}^*.I$ will then uniquely describe a run-time instance of either a read or a write statement. We will also assume that there is a predicate on the size of the w parameter that halts the recursion, causing the function invocation to terminate. If P is a word from $\mathcal{C}^*$, then we define $\text{Current}(P)$ to be the current parameter of the function call. Also, for a control word V we can define $\text{Data}(V)$ to be the path name of the data element actually accessed by that statement, whether

```
F(w) {
          if (|w| > 5) then return
               W         R
     S:   Struct(w) = Struct(w.p) // Read/Write statement
     A:   F(w.a)
     B:   F(w.b);
     C:   F(w.c);
     }
```

Figure 1: Example recursive function

for a read or a write.

There is a total ordering on all the control words obtained from the ordering on the call labels and the statement order within the function. This total order is obtained by extending the order on these symbols to a lexicographic order on the strings of symbols. This ordering corresponds to the actual execution order of the statements for a sequential execution of the program.

The aim of dataflow analysis is, given a particular statement $P.R$, that reads from the structure, to locate the *source*, the write statement that produced that value, should such a statement exist. This characterises the data dependencies within the function. Define two statements U, V as *conflicting* if $\mathrm{Data}(U) \equiv \mathrm{Data}(V)$. We define a *potential source* T as a statement that conflicts with $P.R$ and occurs before it, $T < P.R$. Thus the source is the maximum of all potential sources.

We now consider other work that has been done in both the area of extending the notion of data structures, and applying static analysis to more complex datatypes and control structures.

2 Related Work

2.1 Data Structures

Traditionally, data structures have been based on fixed arrays or trees of values accessed using a pointer based scheme of links. There have been many recent approaches aimed at extending such a system to allow for more complex linkages between different nodes of data.

2.2 Graph Types

The Graph Types approach [12], retains a tree based backbone structure, but augments it with a set of routing directions that can point to a particular node, relative to it. Thus a simple linked list can be converted into a cyclic doubly-linked list. These additional pointers are defined using a *routing expression* which describes the route taken to reach the destination node. It has the disadvantage that the structure must have a tree backbone, and it allows structures where the routing directions depend on the run-time instance of the structure. This makes static analysis more difficult as knowledge not available at compile time would be required to fully analyse such structures.

2.3 Group based fields

Group based fields [8] aims to extend data types to values indexed using a group. Here the directions from a node to a destination node are the generators of the group, the node in direction d from node n is described by the group element $n.d$. Much of the discussion centres on abelian groups which are very similar to arrays, particularly in the kinds of parallelisation that can be applied to them. More general groups are defined using a *finite presentation*, a list of words describing the equality between two particular words, which is extended to the rest of the group. In general the solution of the word problem for a group is an undecidable problem. This makes the idea of analysing programs over such structures intractable, since deciding if two explicitly given nodes are the same is not generally possible. When static analysis is your goal it makes sense to restrict the structures to ones with a unique normal form. It should be mentioned that the group itself is quite a specialised concept, for example all directions must have inverses, which would exclude one way directed graphs.

2.4 ASAP structures

The ASAP approach [11] describes a structure as a set of axioms that hold between certain paths through the structure. An axiom can be one of the three forms:-

$$\begin{aligned} \forall p, p.RE1 &<> P.RE2 \\ \forall p <> q, p.RE1 &<> q.RE2 \\ \forall p, p.RE1 &= p.RE2 \end{aligned}$$

Here RE1 and RE2 are regular expressions over the language of directions. The approach is tailored more towards describing when two nodes are disjoint, rather than providing a unique identity for each node. As a consequence a variety of structures can be described, which can be statically analysed where compile time knowledge is incomplete.

2.5 Static Analysis

We now turn to the problem of performing static analysis of programs written over these general structures. We will be considering recursive functions, and the style will be imperative, rather than functional.

[5] deals with the analysis of array structures, when the program is of the form of a recursive procedure. The procedures consist of sequences of read/write statements or recursive calls. Each call has a recursion guard, a predicate that decides whether the call occurs. Each call is labelled from an alphabet C, so that words over C (*control words*) identify each run-time instance of the function body. The predicates must be computable at compile-time and consist of closed form expressions over the number of occurrences of particular labels in the current control word. Restricting the set of recursion guards in a specific way allows exact predicates for the execution of each statement to be derived.

In [4], some of these restrictions are lifted, and the analysis does the best it can under the assumption that there is no knowledge of whether particular recursive calls are followed.

The data flow analysis aims to find, for a particular statement that reads from a point in the data structure, the last statement that wrote to that particular point. They aim to summarise all the data flow information in closed form formulas, independent of the size of the computation. For this they use numbered occurrence languages.

[3] extends this approach to recursively defined structures. The data structure is defined by describing a language over some alphabet, and then associating each distinct word with an item of data. In this context they consider trees, multidimensional arrays and *almost free shapes*, which are the same as our commuting structures. The recursive program structures that they use are simpler than in [5]. A procedure consists of `explore` constructs that specify a recursion direction and a predicate over the control word. This has the advantage that the body of the procedure is executed if, and only if, the predicates in the `recurse` statements hold. The sources are described using the numbered occurrence languages; for the fully commutative structures the formulas obtained are exact. The free and almost

free shapes, however, yield approximate solutions; a set of possible sources amongst which the actual source must lie.

For the ASAP structures [10] discusses a system that automatically constructs proofs about whether two paths are possible aliases of each other, e.g given $p, q, RE1, RE2$, does $p.RE1 = q.RE2$. This method is suited to general imperative programs where the information obtained about the accesses that each statement makes are of this form. The approach is more appropriate for detecting independent sections of code, rather than tracing the flow of data throughout a section of code.

3 Path naming

As mentioned previously, it is proposed to restrict the class of structures to those that possess a normalising function, and to use that function, possibly expressed as a Markov algorithm, as a description. The idea of a unique path naming strategy is inspired by [9]. It describes a system whereby the nodes of structures are given a name depending on the position of the node. This simplifies the implementation of the data structure for a SPMD machine. Additionally, *aliased* nodes (where nodes can be reached by multiple paths) are dealt with by maintaining a set of alternative names. This contrasts with our approach in that we attempt to study structures where the aliasing can be determined at compile time and the multiple names resolved uniquely. In [9], Gupta does also consider the idea of improving the efficiency of the implementation by analysing the programs.

3.1 Normalisation using Markov algorithms

Markov algorithms are systems with an ordered set of string rewriting rules. Given an input string s, the algorithm searches through the set, and when a rule is matched then s is updated to s' by applying the rule. This procedure is repeated on s' by returning to the first rule. The loop is terminated either when the process is *blocked* (*i.e.* no rules can be applied), or when one of a set of marked *termination* rules is applied. In our case, we accept blocking as termination and ignore the use of explicit termination rules. Markov algorithms have been chosen because:-

- They are sufficiently powerful to describe all string processing algorithms.
- The example data structures considered so far can be simply expressed using them.

$$
\begin{array}{rcl}
a + a^{-1} & \rightarrow & \epsilon \\
b + b^{-1} & \rightarrow & \epsilon \\
c + c^{-1} & \rightarrow & \epsilon \\
b + a & \rightarrow & +ab \\
c + a & \rightarrow & +ac \\
a + b & \rightarrow & +ba \\
-ba & \rightarrow & -ab \\
a + c & \rightarrow & +ca \\
-ca & \rightarrow & -ac \\
+ & \rightarrow & - \\
- & \rightarrow & \epsilon
\end{array}
$$

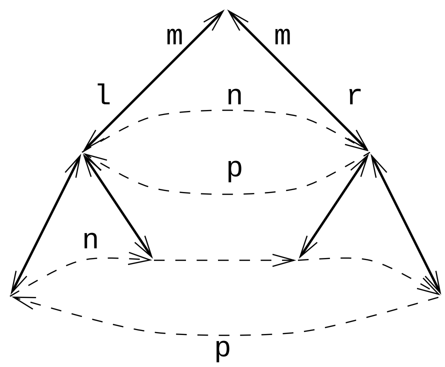
Figure 2: Markov Algorithm for the example commuting/inverse structure

- The commuting/inverse family of structures can have their Markov normalising algorithms automatically generated from their specifications.

As mentioned above, we can generate the Markov algorithm for the path normalisation automatically, given the list of the commuting and inverse directions. Consider a simple example where we have three directions $\{a, b, c\}$, their inverses $\{a^{-1}, b^{-1}, c^{-1}\}$, and directions a and b commute (the algorithm is given in figure 2). The '$+$' and '$-$' are auxiliary symbols which are not in the language of the data structure, and mark the two phases of the algorithm. The algorithm assumes that the path name which is being normalised is of the form $p + x$, where p is an already normalised path name and x is the new direction being added. It actually finds the minimum path with respect to an order derived from an ordering on the alphabet D, identical to the concept of a *geodesic*, see [6].

The set of rules can be built automatically from the description of the commuting directions in the following way. For each direction a we add the following set of rules:-

$$
\begin{array}{rcll}
a + a^{-1} & \rightarrow & \epsilon & \\
x + a & \rightarrow & +ax & \forall x. \quad x \sim a
\end{array}
$$



$$\begin{aligned} ln &\rightarrow r \\ rn &\rightarrow nl \\ n &\rightarrow \epsilon \\ lp &\rightarrow pr \\ rp &\rightarrow l \\ p &\rightarrow \epsilon \\ lm &\rightarrow \epsilon \\ rm &\rightarrow \epsilon \\ m &\rightarrow \epsilon \end{aligned}$$

Figure 3: Markov algorithm for illustrated binary tree

$$-ax \quad \rightarrow \quad x - a \qquad \forall x. \quad x \sim a \quad \wedge \quad x < a$$

We then append two extra rules: to switch between the two phases of the algorithm and to remove the '$-$' symbol at the end of the normalisation.

$$\begin{aligned} + &\rightarrow - \\ - &\rightarrow \epsilon \end{aligned}$$

3.2 Another example structure: A modified binary tree

Here we look at an example of a binary tree with additional pointers. Apart from the main l and r directions, we have a parent pointer, m and the nodes on each level are joined into a doubly linked cyclic list, with n and

p directions. The algorithm for normalisation is shown in figure 3. Note that we can always travel in all directions from all nodes, the parent of the root node is defined as the root node, and the n and p fields wrap around to create a cyclic list.

4 The Static Analysis

We outline in general terms the process of performing the dataflow analysis for the commuting/inverse structures. For the sake of brevity, we have one read and one write in the function - R reads from $v.r$ and W writes to $v.w$, where v is the current path name parameter of the function. We could extend the analysis to any number of reads and writes by repeating this process for each pair. We would combine the results by taking the maximum of this set of sources. The approach is close to that found in [7].

1. We start with a general control word $V.R$, that reads from some part of the structure. Any inverse directions in r or w can be used to restrict the structure of word V. The source can be ignored should Data($V.W$) or Data($V.R$) be illegal path names, as the statement will not be executed anyway.

2. We form the path name $v.r$ and convert it back into a control word $V'.W$, where V' satisfies Data($V'.W$) $\equiv$ Data($V.R$) and $V' < V$. We have the additional constraint that Data($V'.R$) must be legal. V' is the potential source.

3. We can now manipulate the control word V', using the analogous commutative properties from the data structure. We need to maximise it subject to $V'.W < V.R$. This last stage is strongly related to the *minimisation* of the path name in the normalisation algorithm.

The second step of finding the potential source is in general the most difficult. However, restricting the read and write words to simple directions simplifies this stage significantly, as will be shown in the worked example in section 5. During the analysis legality constraints are imposed on the tail of the control word. The lemmas in the appendix allow us to ignore the head of the word and find the sources of the relative control word, then extend this result to a general one, by prepending any control word.

5 Example data flow analysis

Here we will demonstrate the analysis for a particular restricted set of programs. The restrictions will be as follows:-

- Use structures with commuting/inverse directions.
- Restrict all read/write words to single letter words (directions and inverses), plus the empty word.
- All statements occur before all recursive calls in the function body.

We analyse each pair of statements in turn. The statements have the following format, where v is the current word, and we denote the write part of S_i as W_i and the read R_i.

$$\begin{aligned} S_1: \quad & (Struct)(v.w_1) = (Struct)(v.r_1) \\ S_2: \quad & (Struct)(v.w_2) = (Struct)(v.r_2) \end{aligned}$$

With reference to this pair of statements, we now want to find, for any control word of the format $P.R_2$, the corresponding source of the form $P'.W_1$, if one exists. The actual source of the read would then be found by repeating the procedure for all write statements, and taking the latest one. We need to check for the legality of the path in order to ensure that the statements are actually executed.

Since it restricts the solution the most, we only consider the situation where r_i are inverses and w_i are straightforward directions, and are all distinct. Only r_1, r_2 and w_1 affect the legality of the data word, we must have directions r_1^{-1}, r_2^{-1} and w_1 in the current data word before the current statement and its potential source will be executed. For readability we will rename these directions a, b, c, respectively. We also label the recursive calls in the same directions as A, B, C. This gives us a correspondence between an unnormalised path name and the control words that access them; we will use the two words interchangeably, and speak of A and B commuting, when it is actually a and b which do.

Firstly, in order to cancel the b^{-1} inverse, we must be able to commute an existing b within the word to the end. Having cancelled it, for the source to be of the correct form, we must be able to cancel the a^{-1}, and move the c to the end of the path for the write part of the statement. This gives us that $a \sim c$; if not then S_1 can be disregarded as a potential source. Let us define the set of directions that commute with A as $\mathrm{Com}(A)$, P can be one of six forms, depending on the relative position of the last A, B, C in the word. Some of the forms place additional restrictions on the commuting properties of the directions in order for the paths accessed to be legal.

$$
\begin{aligned}
&X.A.X_1.B.X_2.C.X_3 && b \sim c && (1)\\
&X.A.X_1.C.X_2.B.X_3 && && (2)\\
&X.B.X_1.A.X_2.C.X_3 && b \sim a, b \sim c && (3)\\
&X.B.X_1.C.X_2.A.X_3 && b \sim a, b \sim c && (4)\\
&X.C.X_1.A.X_2.B.X_3 && && (5)\\
&X.C.X_1.B.X_2.A.X_3 && b \sim a && (6)
\end{aligned}
$$

The X_i are composed of directions with particular commuting properties. For example, in the first, we have $X_1 \in \mathrm{Com}(A)^*$, $X_2 \in (\mathrm{Com}(A) \cap \mathrm{Com}(B))^*$ and $X_2 \in (\mathrm{Com}(A) \cap \mathrm{Com}(B) \cap \mathrm{Com}(C))^*$. For a given set of commuting properties we must consider each of the possible forms and produce a source for each. Additional constraints on the order of the statements can produce a description of the potential source that relies only on the structure of the word as given above. For example, suppose out of a, b, c, only directions a and c commute. This implies that the path is of the form of (2) or (5). If it is of the form of (2) then the corresponding read and write words will be:

$$
\begin{aligned}
&X.A.X_1 \quad .C.X_2.B.X_3.R && (7)\\
&X.A.X_1 \quad .X_2.X_3.W && (8)
\end{aligned}
$$

Now provided $C > X_1$, $C > X_2$ and $C > W$, the write will be a potential source. Similarly, for the second form we have the two control words:

$$
\begin{aligned}
&X. \quad C.X_1.A.X_2.B.X_3.R && (9)\\
&X. \quad X_1.A.X_2.X_3.W && (10)
\end{aligned}
$$

This gives us a potential source provided that $C > X_1$ and $C > A$. For some of the other combinations, the form of the potential source depends on the actual structure of the sections of the control word, so describing the general solution is a more complex task.

6 A binary tree

Since we can simply define an inverse for each direction, finding the source of a read is simplified. For a pair of read/write statements with directions r and w, we apply r to the current path of the particular statement and then apply w^{-1} to find the potential source. We can then check that it occurs before the read, and reject it otherwise. We take a simple example:-

```
F(w) {
     S:   Struct(w.m) = ...
     T:   ...   = Struct(w.p)
     L:   F(w.l);
     R:   F(w.r);
     }
```

Take a typical control word that reads in statement T, denoted $X.T$. For the purpose of applying direction p, X can have two formats, depending on whether it contains an R. These are $X'.R.L^n$ or L^n, where X' is any control word and $n \geq 0$.

We take the L^n form first, we now want to apply p to it to find the current word of the path name that it accesses. This gives R^n. We now apply the inverse of the write word m to find the control word that accesses that part of the structure. There are two possible inverses - either append an L or a R. The two possible sources of $L^n.T$ are $R^{n+1}.S$ and $R^n.L.S$. Both of these control words are executed after the read, so this form has no source. If all the possible writes produce this result then this may indicate that the program is using an uninitialised value.

Now consider $X'.R.L^n.T$. The current word of the path that it accesses is now $X'.L.R^n$. Applying the inverse of the write word leaves us with possible sources of $X'.L.R^n.L.S$ and $X'.L.R^{n+1}.S$. Both of these occur earlier than the read. However the second is always executed after the first. Therefore it is the source.

It is possible to see how this analysis could be extended to cover read and write words rather than just single directions.

7 Future work

In order to generalise the existing approach the following problems will need to be considered.

1. Extend data structures. The approach so far has been based on specific examples. In order to be of wider value, a more general framework needs to be used.

    (a) Work with the group fields and graph types frameworks to extend dataflow analysis to these ideas. These methods will need some modification, since the graph types system does allow structures with runtime dependencies, and the group approach is possibly too general to allow accurate analysis.

    (b) Extend existing approach to a collection of properties, *e.g* commuting, inverse, parent, *etc* ... , allowing them to be combined to produce more general families of structures.

2. Extend program model. One of the main restrictions placed on the expressiveness of the functions is the lack of general predicates on the recursive calls. There are three methods worth considering.

    (a) Follow 'fuzzy' approach. If we do not want to restrict the form of such predicates, or allow them to be runtime dependant, we need to follow the ideas in [4] and [1]. The approach would then be to do the best possible with no knowledge of which predicates are evaluated as true. This means to locate the set of all potential sources, and prune away the earlier ones whose execution implies the execution of a later potential source.

    (b) Follow group fields technique, define a subset of the potentially infinite structure that the program operates on. This is similar to ideas in [8], where a 'formating language' specifies the structure of allowable path names. The format used is similar to the numbered occurrence languages used by [5]. This approach obviously restricts the expressiveness of the program, but allows for more accuracy in the solutions.

    (c) Use recursive `constructs` as in [3] to build up a the structure first. The functions then only operate over the existing values. This is similar to the previous approach, but can allow for more complex sub-sets of the structure to be selected.

To summarise, there are many existing approaches to extending the description and manipulation of data structures. Not all of these are geared towards the kind of compile-time analysis that is required to parallelise sequentially written code. The ASAP techniques do address this problem but aim primarily for alias information rather than dataflow analysis. The difference between these is that alias information will discover that two statements access the same item of data, when in fact there may be no true dependence, since the value is overwritten between them. The dataflow approach has the advantage that potentially more independence can be found when the full dataflow of the code is known. The disadvantage is of course that the analysis is much harder, and impossible if insufficient information is available. Inspired by the analysis that can be achieved when 'static' structures such as arrays are manipulated, my research will aim to find a more general class of structures for which this quality of analysis is possible. The disadvantage of this is that in many problems the data structures do depend on some run time information. To be more widely applicable my solution will aim to compliment these other methods.

References

[1] Denis Barthou and Jean-Francois Collard. Fuzzy array dataflow analysis. *Journal of Parallel and Distributed Computing*, 40(2):210–241, February 1997.

[2] A. T. Berztiss. *Data Structures - Theory and Practice*. New York : Academic Press, 1971.

[3] A. Cohen, J.-F. Collard, and M. Griebl. Data flow analysis of recursive structures. Technical Report 96/018, Laboratory PRiSM, University of Versailles, France, September 1996.

[4] Albert Cohen and Jean-Francois Collard. Fuzzy array data-flow analysis – part II: Recursive programs. Technical Report 96-036, PRiSM, `http://www.prism.uvsq.fr`, December 1996.

[5] Albert Cohen, Jean-Francois Collard, and Martin Griebl. Array dataflow analysis for imperative recursive programs. Technical Report 96/035, Laboratory PRiSM, University of Versailles, France, 1996.

[6] D. B. A. Epstein. *Word Processing in Groups*. Addison Wesley, 1992.

[7] P Feutrier. Dataflow analysis of scalar and array references. *International Journal of Parallel Programming*, 20(1):23–53, February 1991.

[8] J Giavitto, O Michel, and J. P Sansonnet. Group based fields. In *Parallel Symbolic Languages and Systems, International Workshop PSLS*, 1995.

[9] Rajiv Gupta. SPMD execution of programs with pointer-based data structures on distributed memory machines. In *Journal of Parallel and Distributed Computing*, volume 16, pages 92–107, 1992.

[10] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A general data dependence test for dynamic pointer-based data structures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 100–104, June 1994.

[11] Joseph Hummel, Laurie J Hendren, and Alexandru Nicolau. A language for conveying the aliasing properties of dynamic, pointer-based data structures. In *Proceedings of the 8th International Parallel Processing Symposium*, April 1994.

[12] Nils Klarlund and Michael I. Schwartzbach. Graph types. In *Proceedings of the ACM 20th Symposium on Principles of Programming Languages*, pages 196–205, January 1993.

Appendix

As mentioned earlier, we want to calculate the sources *relative* to a particular control word. The following lemmas allow us to do this. Note that these apply only to the commuting/inverse structures, they are not general properties of structures with an equivalence relation.

Lemma 1 *For structures with commuting and inverse directions, if u and v are legal (contain no inverse directions) then $u \equiv v \iff x.u \equiv x.v$ where $x, u, v \in D^*$.*

Proof: $\Rightarrow$ follows directly from the fact that the equivalence relation is of the form of a set of string rewriting rules. $\Leftarrow$ does depend on the nature of the rules. Let us suppose $x.u \equiv x.v$. Since the rules only allow for the transposition of commuting directions, and $x = x$, we infer that u and v must contain the same set of letters. Now, if u and v only contain commuting directions then we have the result. Let us assume otherwise, that there is a pair of non-commuting directions in u. We observe that in $x.u$ any non-commuting pairs must retain the same strict order when we apply rules to convert it to $x.v$. This same order will be preserved when we compare u and v. Therefore all non-commuting pairs in u and v will retain the same relative order, thus $u \equiv v$.

Definition 1 *Two control words V and V'* conflict*, if one reads a value that the other writes to,* i.e. *$Data(V.R) \equiv Data(V'.W)$.*

Lemma 2 *If V and V' conflict then, $\forall X \in C^*$, $X.V$ and $X.V'$ conflict.*

Proof: Follows from Lemma 1, since $\mathrm{Data}(A.B) \equiv \mathrm{Data}(A).\mathrm{Data}(B)$.

Lemma 3 *If $X'.W$ is the source of $X.R$, then $\forall V \in C^*$, if $V.X'.W$ is executed then it is the source of $V.X.R$.*

Proof: $X'.W$ being the source of $X.R$ means that they conflict and $X'.W < X.R$, and if $Y.W$ and $X.R$ conflict, for any $Y \in C^*$, then $Y.W < X'.W$.

Firstly $V.X'.W < V.X.R$ and $V.X'.W$ conflicts with $V.X.R$, from Lemma 2. Now suppose that there exists a Z, such that $V.X'.W < Z.W < V.X.R$, and $Z.W$ conflicts with $V.X.R$. The inequalities imply that Z is of the form $V.K$ for some K. Since $Z = V.K$ conflicts with $V.X$, K must conflict with X, from Lemma 2. This implies that $K.W$ should be the source of $X.W$, which is a contradiction, hence the result.