Formal Development of a Minimal Register Allocation Algorithm

Kent Lee Hantao Zhang
Department of Computer Science
The University of Iowa
Iowa City, IA 52242
kent-lee@uiowa.edu
hantao-zhang@uiowa.edu

May 13, 1999

Abstract

Register allocation is a part of the code generation phase of a compiler where symbolic registers are assigned physical registers of a particular machine architecture. Classically, register allocation has been assumed to be equivalent to graph coloring, which is an NP-Complete problem. With the aid of an algebraic specification system, we develop a new linear time algorithm for minimal register allocation. In this paper, we show how the algorithm is algebraically specified and formally verified. The algorithm works on register interference graphs and we develop a data structure called α-lists for efficiently representing register interference graphs.

1 Introduction

Aho, Sethi, and Ullman [ASU86] define a compiler as "... a program that reads a program written in one language – the source language – and translates it into an equivalent program in another language – the target language ...". A compiler typically accomplishes this task in phases which are usually described as scanning, parsing, and code generation.

When the target language is the assembly or machine language of a particular machine, one part of the code generation phase is register allocation. Code generation can be simplified if the assumption can be made that the target machine has an unlimited number of registers. This is especially true for pools of similar registers such as general purpose registers or floating point registers. These unlimited sets of registers are referred to as *symbolic registers*, while the actual registers of the machine are referred to as *physical registers*.

A new symbolic register is created each time the code generator requests a physical register of the target machine. When a symbolic register is no longer needed by the computation being compiled, it is discarded and not used again. In other words, a symbolic register is *alive* from the time it is created until it is discarded. Then, after code generation has completed, each symbolic register is assigned to a physical register so that no two symbolic registers that are alive at the same time share the same physical register. A more formal definition of the problem follows.

> **Definition**: A **register interference graph** is a simple undirected graph, $G = (V, E)$, where $V = \{s_1, s_2, ..., s_n\}$ is a set of symbolic registers and E is a set of edges where $(s_i, s_j) \in E$ iff s_i and s_j are *alive* at the same time.

Register allocation and graph coloring are obviously closely related. That is, if we can use k colors to color a register interfence graph such that no two adjacent vertices have the same color, then at most k physical registers are needed for these n symbolic registers. In fact many texts on compiler construction claim they are equivalent. Aho, Sethi, and Ullman [ASU86] give heuristics that can be applied to a register interference graph when attempting to color it in a given number of colors. Pittman and Peters [PP92] claim that they are equivalent but also say "we have found an informed demand allocation scheme to give good results without the complexities of graph coloring...".

Since graph coloring is an NP-complete problem, the equivalence of register allocation and graph coloring would imply that register allocation is an NP-complete problem. However, the register interference graphs we consider here are not arbitrary graphs. These graphs possess a special property called the *lifetime property*. By exploring this property, we are able to design an $O(|V|+|E|)$ time algorithm to solve the minimal register allocation problem.

During the development of the new algorithm, we heavily relied on some algebraic tools such as RRL (an inductive theorem prover for algebraic specifications [KZ89]). We used RRL to test the algorithm and more significantly, we used RRL to formally prove the correctness of the algorithm.

2 A New Register Allocation Algorithm

2.1 An Example

h:=(a + (b * c)) + (d + (e * f))

ld	s_1,	a
ld	s_2,	b
ld	s_3,	c
mul	s_2,	s_3
add	s_1,	s_2
ld	s_4,	d
ld	s_5,	e
ld	s_6,	f
mul	s_5,	s_6
add	s_4,	s_5
add	s_1,	s_4
st	s_1,	h

Figure 1: An example of source and target codes

We first provide an example to illustrate the involved concepts. Consider the assignment statement and its equivalent target language construct in Figure 1. Assume that the target language is the assembly language of a simple load/store architecture and the first operand is the destination of the result for arithmetic operations (e.g. `mul a, b` is $a := a * b$). Figure 2 is the register interference graph associated with the target code in Figure 1.

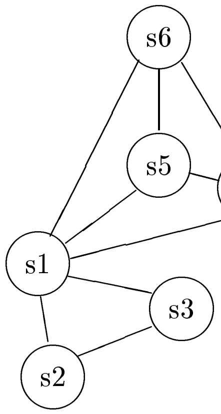

Figure 2: A register interference graph

Clearly, the target code in Figure 1 could be optimized, producing a much simpler register interference graph. However, optimization is a separate problem and this example works nicely for our purposes. Another problem to consider is what should be done if a target language computation requires more physical registers than exist in the machine. Then, register spilling is required. Again, this is a separate problem which is not considered here. But, we would like to know the minimal number of physical registers required for a target language computation so we can decide if spilling is required. For the example, at least four physical registers are needed. A mapping from the symbolic register set to the physical register set that satisfies the constraints of the register interference graph in Figure 2 is $\{s_1 \rightarrow p_1, s_2 \rightarrow p_2, s_3 \rightarrow p_3, s_4 \rightarrow p_2, s_5 \rightarrow p_3, s_6 \rightarrow p_4\}$.

2.2 The Lifetime Property and a Major Theorem

It turns out that the key to solving the register allocation problem is in the lifetime property. This property was implied above when we stated that symbolic registers are created and then discarded and not used again. That is, they are born when they are created, and die when they are discarded. In other words, they have one lifetime. This property can be stated more mathematically.

> **Definition**: A simple undirected graph $G = (V, E)$, exhibits the **lifetime property** iff $\exists \pi : N \rightarrow V$ so that if $(\pi(i), \pi(k)) \in E$, then $(\pi(i), \pi(j)) \in E$, $i < j < k$ where π is a one-to-one mapping for $N = \{1, 2, ..., |V|\}$.

Note that finding this mapping π, if it exists, for a graph would take exponential time in general, so this does not provide a polynomial time algorithm to solve the graph coloring problem for all graphs for which the lifetime property holds. However, in the case of register interference graphs, the mapping is ready to obtain. It is simply the order in which the symbolic registers were requested. For simplicity, we assume from now on that the mapping π is implicitly given as the indices of vertices in V for any graphs with the lifetime property, that is, $\pi(i) = v_i$ for any $v_i \in V$.

In the following, we propose a new algorithm for minimally coloring a simple undirected graph $G = (V, E)$ with the lifetime property. The algorithm is implicitly given in the proof of the following theorem, whose proof is done by induction on the number of vertices in V.

> **Major Theorem**: A simple undirected graph, $G = (V, E)$ exhibiting the lifetime property can be minimally colored in time $O(|V| + |E|)$.
>
> **Proof**: (by induction on the cardinality of V)
>
> **Base case**: $|V| = 0$. Clearly, G can be colored by 0 colors in constant time.
>
> **Inductive case**: Assume, by the induction hypothesis, for all $G = (V, E)$ with $|V| = n$ that G can be minimally colored by k colors in time $O(|E| + |V|)$ if G exhibits the lifetime property. Consider $G' = (V \cup \{v_{n+1}\}, E \cup E')$ with $|V \cup \{v_{n+1}\}| = n + 1$ where E' contain only the edges from vertices in V to the vertex

v_{n+1}. We must consider three cases for G', which is supposed to exhibit the lifetime property.

Case 1: v_{n+1} has less than k neighbors. Clearly, v_{n+1} can be colored one of the k colors of G that is not one of the colors of the neighbors of v_{n+1}. The neighbor colors of v_{n+1} can be determined by examining the coloring for G and the neighbors of v_{n+1} in E'; this can be done in time $O(|E'|)$. Since no new color is needed, G' is still minimally colored.

Case 2: v_{n+1} has k neighbors. Then, since the lifetime property holds for G', if $(v_i, v_{n+1}) \in E'$ then $(v_i, v_j) \in E$, for $i < j < n+1$. This means that v_{n+1} forms a $(k+1)$-clique with its k neighbors in V. But we know from graph theory that a simple undirected graph with a $(k+1)$-clique cannot be colored in less than $k+1$ colors. Therefore, we have to assign a new color to v_{n+1}; G' is thus still minimally colored with $k+1$ colors. This can be done again in time $O(|E'|)$.

Case 3: v_{n+1} has more than k neighbors. Then there exists at least a $(k+1)$-clique in G by the same argument as case 2. But this means that G cannot be colored with k colors. This contradicts our induction hypothesis, so this case is invalid.

Therefore, G' can be minimally colored if G is minimally colored, and we can conclude by induction that all simple undirected graphs exhibiting the lifetime property may be minimally colored in time $O(|V| + |E|)$ (assuming we know the mapping of the vertices of V). $\square$

The hand proof given here is the central idea exploited in both a formal specification of the algorithm given below and the correctness proof in the next section.

2.3 An Algebraic Specification of the Algorithm

To design any algorithm, we must first choose a good data structure. A graph is a rather complex object. It requires the representation of two sets, a set of vertices and a set of edges. However, we have invented a rather simple and eloquent representation for register interference graphs. That is, a register interference graph can simply be represented as a list of natural numbers. However, the algorithm based on this data structure will have complexity $O(|V|^2)$ instead of $O(|V| + |E|)$.

For a register interference graph $G = (V, E)$, where $V = \{v_1, v_2, ..., v_n\}$, we know by the lifetime property that if $(v_i, v_k) \in E$, then $(v_i, v_j) \in E$ for $i < j < k$. It turns out, for the purposes of the algorithm, to work best if we reverse the order of the vertex indices. That is, let V be listed as u_0, u_1, ..., u_{n-1}, where $v_j = u_{n-j}$ for $1 \leq j \leq n$.

Let $\alpha(u_i) = |\{(u_k, u_i) \in E \mid k < i\}|$. It is easy to check that G can be reconstructed from $\alpha(u_i)$ where $0 \leq i < n$. Hence, G can be represented as a list of integers

$$[\alpha(u_0), \alpha(u_1), ..., \alpha(u_{n-1})].$$

Let us call this list the α-**list** of the graph.

So, for instance, the graph shown in Figure 2 can be represented by the α-list $[0, 1, 2, 0, 1, 5]$. For a list of integers to be a valid α-list of a register interference graph, no number in the list should be greater than its position, i.e., $\alpha(u_i) \leq i$. So, $\alpha(u_0)$ should always be 0, $\alpha(u_1)$ should 0 or 1, $\alpha(u_2)$ should be 0 or 1 or 2, and so on. However, because the algorithm's functions are all total, this constraint isn't needed. That is, if $\alpha(u_i) > i$, it is assumed that the vertex u_i has edges to all the vertices u_j for $0 \leq j < i$.

Using α-lists, it is also easy to decide if two vertices are neighbors in G.

> **Lemma:** For any $i < j$, $(u_j, u_i) \in E$ iff $\alpha(j) \geq j - i$.

In our algorithm, the color assigned to vertex u_i is represented by a number $c(u_i)$, $0 \leq c(u_i) < n$. For the vertices u_0, u_1, ..., u_{n-1}, we use the color list $[c(u_0), c(u_1), ..., c(u_{n-1})]$.

Given below is a formal specification of the algorithm as described in the proof of the previous theorem. It is written in a slightly modified RRL syntax [KZ89]; the same specifications have been converted into Standard ML for testing many examples.

Assume that the data type `list` denotes lists of natural numbers. Both the α-list and the color list are represented as list of natural numbers. For readability, we use G for the α-list and C for the color list. The formula `if3(x,y,z)` is equivalent to `if x then y else z`.

```
func color : list -> list  ;;; the recursive algorithm for coloring
color(nil) == nil
color(cons(x,G)) ==
    cons(firstnotin(sort(neighborcolors(G, color(G), 0)), 0), color(G))

func neighborcolors : list, list, nat -> list
  ;;; neighborcolors(G,C,0) returns the colors of the neighbors of u0.
neighborcolors(nil, C, z) == nil
neighborcolors(G, nil, z) == nil
neighborcolors(cons(x,G), cons(y,C), z) ==
    if3(z < x, cons(y, neighborcolors(G, C, z+1)),
                       neighborcolors(G, C, z+1))

func firstnotin : list, nat -> nat
  ;;; firstnotin(x, z) returns the minimal number >= z but not in list x,
  ;;; where x is a nondecreasing list of integers.
firstnotin(nil, z) == z
firstnotin(cons(x, y), z) == if3(z < x, z, firstnotin(y, z+1))
```

The reader is encouraged to compare the main function *color* with the proof of the major theorem. That is, let $G' = (V \cup \{u_0\}, E \cup E')$, then $color(G')$ returns a list of colors by (i) calling $color(G)$, where $G = (V, E)$; (ii) collecting the colors used by the neighbors of u_0; and (iii) choosing the minimal unused color by the neighbors for u_0.

Step (i) is obvious. Step (ii) is implemented by calling function *neighborcolors*: $neighborcolors(G, C, 0)$ returns a list of the colors of the neighbors of vertex u_0, where G and C are, respectively the α-list and the color list of u_1, u_2, ..., u_{n-1}. The condition $z < x$ tests if the current vertex is a neighbor of u_0 according to the above lemma.

Step (iii) is implemented by function *firstnotin* and *sort*: $sort(x)$ sorts x into a nondecreasing list of integers; for the implementation, we used the insert-sort algorithm. Function $firstnotin(x, z)$ returns the minimal number not less than z but not in x.

3 Proof of Correctness

The algorithm given in the previous section has been checked for all register interference graphs of size one vertex up to eight vertices. However, we would like to be able to prove its correctness for all register interference graphs. We have used the automated inductive theorem prover of RRL [KZ89] to achieve this goal.

RRL (*Rewrite Rule Laboratory*) [KZ88, KZ89] was originally developed as an environment for experimenting with automated reasoning algorithms for equational logic based on rewrite techniques. It has now matured into a full-fledged theorem prover which has been used to solve hard and challenging mathematical problems in automated reasoning literature as well as a research tool for investigating the use of formal methods in hardware and software design. We have used RRL for software verification [ZGH91, ZH92] and hardware verification [HZ93].

The algebraic specification of the algorithm is unchanged from the previous section. To formally verify the correctness of the algorithm, we have to check two properties:

1. The list $[c_0, ..., c_{n-1}]$ returned by $color(G)$ is a proper coloring of the vertices $u_0, u_1, ..., u_{n_1}$. That is, whenever $(u_i, u_j) \in E$, $c_i \neq c_j$.
2. The value $1 + max\{c_0, ..., c_{n-1}\}$, i.e., the number of colors used by $color(G)$, should be minimal. One is added because zero is one of the used colors.

RRL is able to prove both properties automatically after the user supplies a list of auxiliary lemmas (each of which is first proved to be correct before they are used to prove others). Since the proofs involve many techniques implemented in RRL which cannot be covered here, we simply show how to specify the above two properties without giving the details of the proofs. A list of RRL commands for fulfilling the proof is provided in the appendix of the paper.

3.1 It Produces a Coloring of G

First, we define a predicate *coloring*: $coloring(G, C)$ returns true iff C is a proper coloring of G, where G is the α-list and C is the color list.

```
func coloring : list, list -> bool
coloring(nil, C) == true
coloring(G, nil) == true
coloring(cons(x, G), cons(y, C)) ==
   not member(y, neighborcolors(G, C, 0)) and coloring(G, C)
```

This is our first major theorem. It states the call $color(G)$ produces a correct coloring for all register interference graphs, G. This follows automatically from several proved lemmas and the axioms (i.e., the definitions).

```
coloring(G, color(G)) == true
```

This is a fairly easy theorem for RRL to prove.

3.2 It Produces a Minimal Coloring

To show that the number of colors used by $color(G)$ is minimal, we prove that $1 + maxelem(color(G))$, where $maxelem(x)$ returns the maximal element in list x, is no greater than the maximal clique number in G.

By the lifetime property of register interference graphs, the maximal clique of G containing vertex u_0 is one plus the number of neighbors of u_0 (please refer to the proof of the major theorem in section 2).

The following functions define the function *maxclique*.

```
func neighbors : list, nat -> nat
   ;;; count the number of neighbors of u0.
neighbors(nil, z) == 0
```

```
neighbors(cons(x, G), z) ==
   if3(z < x, neighbors(G, z+1)+1, neighbors(G, z+1))

func maxclique1 : list -> nat
   ;;; return the maximal clique minus 1.
maxclique1(nil) == 0
maxclique1(cons(x, G)) == max(neighbors(y, 0), maxclique1(G))

func maxclique : list -> nat
   ;;; maxclique(G) returns the maximal clique number of G.
maxclique(G) == maxclique1(G) + 1

func num_colors : list -> nat
   ;;; num_colors(G) returns the number of colors used by color(G).
num_colors(G) == maxelem(color(G)) + 1
```

The intended property for RRL to prove is: $\mathit{num\_colors}(G) \leq \mathit{maxclique}(G)$. After proving 8 additional auxiliary lemmas, RRL is able to prove this property automatically in less than 30 seconds on a HP 715 workstation (running the AKCL lisp). Since any graph cannot be colored in less colors than the maximum clique number, we conclude from this property that the coloring is indeed minimal.

4 Conclusion and Future Work

We have shown that contrary to popular belief, register allocation and graph coloring are not equivalent problems. This is based on the fact that register interference graphs may be constructed so they are restricted by the lifetime property. Therefore, because of the lifetime property, a linear time algorithm exists for the minimal coloring of register interference graphs. We have provided an algebraic specification of the algorithm using a compact data structure for register interference graphs. We have also proven the correctness of this algorithm using the RRL theorem prover. Our experiences indicate that algebraic tools like RRL are very useful for formal software development.

Acknowledgements

We would like to thank Tom Halverson, Sriram Pemmaraju, Ken Slonneger, and Eric Van Wyk for their help in preparing this paper.

References

[ASU86] Aho, Sethi, and Ullman: Compilers: Principles, Techniques, and Tools (Addison Wesley), 1986.

[HZ93] Hua, G.X., Zhang, H.: (1993) Formal verification of circuit designs in VHDL. In the Proc. of 9th International Conference on Advanced Science and Technology, Motorola, Inc., Schaumburg, Illinois.

[KZ88] Kapur, D., and Zhang, H.: RRL: a Rewrite Rule Laboratory. In: Proc. *Ninth International Conference on Automated Deduction*, E. Lusk and R. Overbeek (eds.), Lecture Notes in Computer Science, Vol. 310, Springer-Verlag, Berlin, 1988, pp. 768–769.

[KZ89] Kapur, D., and Zhang, H., An overview of RRL: Rewrite Rule Laboratory. In: Proc. *Third International Conference on Rewriting Techniques and Its Applications*, Lecture Notes in Computer Science, Vol. 355, Springer-Verlag, Berlin, 1989, pp. 513–529.

[PP92] Pittman and Peters, The Art of Compiler Design: Theory and Practice (Prentice Hall), 1992.

[ZGH91] Zhang, H., Guha, A., Hua, G.X.: (1991) Using algebraic specifications in Floyd-Hoare assertions. In: Rus, T. (ed.): Proc. of Second International Conference on Algebraic Methodology and Software Technology, Iowa City, Iowa.

[ZH92] Zhang, H., Hua, G.X.: (1992) Proving the Chinese remainder theorem by the cover set induction. In Kapur, D.: (ed.): Proc. of 1992 International Conference of Automated Deduction. pp. 431–445.

Appendix: RRL Input File for Register Allocation

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Definition of nat (Natural numbers)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
input type
  nat = 0 + suc(nat)
end

input axiom
  func 1 : nat
  1 == suc(0)

  func + : nat, nat -> nat
  x + 0 == x
  x + suc(y) == suc(x + y)

  func < : nat, nat -> bool
  x < 0 == false
  0 < (x + 1) == true
  (x + 1) < (y + 1) == x < y

  func >= : nat, nat -> bool
  x >= y == not(x < y)

  func max : nat, nat -> nat
  max(x, y) == if3((x < y), y, x)
end

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Definition of lists of nat
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
input type
  list = nil + cons(nat, list)
end

input axiom
   func maxelem : list -> nat    ;; find the maximum number in a list of numbers
   maxelem(nil) == 0
   maxelem(cons(x, y)) == max(x, maxelem(y))

   func length : list -> nat
   length(nil) == 0
   length(cons(x, y)) == length(y) + 1

   func notin : nat, list -> bool
   notin(x, nil) == true
   notin(x, cons(y, z)) == not(x = y) and notin(x, z)

   func distinct : list -> bool
   ;; distinct(xlist) returns true iff all elements of xlist are distinct.
   ;; A set is represented by a list of distinct elements.
   distinct(nil) == true
```

```
   distinct(cons(x, y)) == notin(x, y) and distinct(y)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Functions for defining insert-sort.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
   func insert : nat, list -> list
   insert(x, nil) == cons(x, nil)
   insert(x, cons(y, z)) ==  if3(x = y, cons(y, z),
            if3(x < y, cons(x, cons(y, z)), cons(y, insert(x, z))))

   func sort : list -> list
   sort(nil) == nil
   sort(cons(x, y)) == insert(x, sort(y))

   func sortedp : list -> bool
   sortedp(nil)  == true
   sortedp(cons(xa, nil))  == true
   sortedp(cons(xa, cons(ya, x)))  == if3(ya < xa, false, sortedp(cons(ya, x)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Functions for defining maxclique
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
   func neighbors : list, nat -> nat  ;;; count the number of neighbors of u0.
   neighbors(nil, z) == 0
   neighbors(cons(x, G), z) ==
      if3(z < x, neighbors(G, z+1)+1, neighbors(G, z+1))

   func maxclique1 : list -> nat      ;;; return the maximal clique minus 1.
   maxclique1(nil) == 0
   maxclique1(cons(x, G)) == max(neighbors(y, 0), maxclique1(G))

   func maxclique : list -> nat       ;;; return the maximal clique number.
   maxclique(G) == maxclique1(G) + 1

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Functions for defining color
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
   func firstnotin : list, nat -> nat
   ;;; firstnotin(x, z) returns the minimal number >= z but not in list x,
   ;;; where x is a nondecreasing list of integers.
   firstnotin(nil, z) == z
   firstnotin(cons(x, y), z) == if3(z < x, z, firstnotin(y, z+1))

   func neighborcolors : list, list, nat -> list
   ;;; neighborcolors(G,C,0) returns the colors of the neighbors of u0.
   neighborcolors(nil, C, z) == nil
   neighborcolors(G, nil, z) == nil
   neighborcolors(cons(x,G), cons(y,C), z) ==
       if3(z < x, cons(y, neighborcolors(G, C, z+1)), neighborcolors(G, C, z+1))

   func newcolor : list, list -> nat     ;;; a shorthand
   newcolor(x, y) == firstnotin(sort(neighborcolors(x, y, 0)), 0)

   func color : list -> list  ;;; the recursive algorithm for coloring
   color(nil) == nil
```

```
   color(cons(x, G)) == cons(newcolor(G, color(G)), color(G))

   func num_colors : list -> nat
   ;;; num_colors(G) returns the number of colors used by color(G).
   num_colors(G) == maxelem(color(G)) + 1

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Functions for defining coloring
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
   func member : nat, list -> bool
   member(x, nil) == false
   member(x, cons(y, z)) == (x = y) or member(x, z)

   func coloring : list, list -> bool
   coloring(nil, C) == true
   coloring(G, nil) == true
   coloring(cons(x, G), cons(y, C)) ==
       not member(y, neighborcolors(G, C, 0)) and coloring(G, C)
end

rule make ;;; make rewrite rules from axioms.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Some Lemmas
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
prove max(x, max(y, z)) == max(max(x, y), z)
prove max(x, x) == x
prove sortedp(sort(x))
prove distinct(sort(y)) if distinct(y)
prove not(firstnotin(y, u) < u)
prove not(firstnotin(y, u) = v) if (v < u)
prove notin(y1, y) == true if (y1 < x) and sortedp(cons(x, y))
prove (u < firstnotin(y, x)) if (u < x)
prove notin(firstnotin(x, y), x) if sortedp(x) and distinct(x)
prove notin(x, sort(y)) == notin(x, y) if distinct(y)
prove notin(x, y) if notin(x, sort(y))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Major Theorem 1
prove coloring(G, color(G)) == true
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

prove (x >= y) if (x >= z) and (z >= y)
prove (length(x) + y) >= firstnotin(x, y)
prove length(sort(x)) == length(x)
prove length(color(x)) == length(x)
prove length(neighborcolors(x, y, u)) == neighbors(x, u) if length(x) = length(y)
prove length(neighborcolors(x, y, 0)) < firstnotin(isort(neighborcolors(x, y, 0)), 0) ==
       false if (length(x) = length(y))
prove neighbors(x, 0) < newcolor(x, color(x)) == false
prove (x >= y) if (z < x) and (z >= y)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Major Theorem 2
prove maxclique(G) >= num_colors(G)
```

;;;