A MECHANICAL PROOF OF THE CHURCH-ROSSER THEOREM

N. Shankar

Technical Report 45 March 1985

Institute for Computing Science
2100 Main Building
The University of Texas at Austin
Austin, Texas 78712
(512) 471-1901

The research reported here was supported by National Science Foundation Grant DCR 8202943 and by the Science and Engineering Research Council of United Kingdom

Abstract

This paper presents the highlights of a formalization and proof of the Church-Rosser theorem that was carried out with the Boyer-Moore theorem prover. The Church-Rosser theorem is a celebrated metamathematical result on the Lambda Calculus. It is also of historical interest since there was a gap of more than thirty years between the statement of the theorem and the construction of a widely accepted proof for it. The proof presented in this paper is based on that of Tait and Martin-Lof. The mechanical proof illustrates the effective use of the Boyer-Moore theorem prover in proof-checking difficult metamathematical proofs.

> By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race.
>
> A. N. Whitehead [whitehead]

1. Introduction

The use of computer programs for theorem proving, program verification and proof-checking has been under criticism from various quarters. Most critics have based their comments on misgivings that are grounded in their philosophical attitudes, rather than on any actual experience using such programs. By the same token, proponents of automated theorem proving and program verification have mainly been either users or builders of such systems and can therefore hardly claim objectivity. The purpose of this paper is neither to join in the criticisms nor to respond to them. Rather it is to demonstrate that these computational aids (such as the Boyer-Moore theorem prover) when effectively employed, can be of considerable value in the study and use of formal logical reasoning. They can be used to construct, examine and understand proofs with much greater clarity, rigor and detail than would have been feasible without them.

This paper presents the results of a successful application of the Boyer-Moore theorem prover [boyer] to the proof of the Church-Rosser theorem [church-rosser], a deep metamathematical result on the Lambda Calculus [lambda]. The paper outlines the formalization of this theorem in the Boyer-Moore logic and describes the means by which the theorem prover was led to the proof. In the conclusions, we present some observations regarding the importance of notation in the proof and the role of a high-level automatic proof-checker in the construction and verification of the proof. The paper is self-contained and no familiarity with the Boyer-Moore theorem prover or Lambda Calculus is assumed.

The Lambda Calculus (or λ-calculus) was introduced by Church [barendregt, lambda] in order to study functions as rules of computation rather than as graphs of argument-value pairs. It was hoped that λ-calculus would provide an alternative foundation for logic and mathematics. This aim has remained unfulfilled due to the appearance of contradictions when λ-calculus was extended with logical notions. Lambda calculus has nevertheless been a fruitful medium for the study of functions and computations. The programming language in which the mechanical proof was formalized, a variant of pure-Lisp [mccarthy], was one of the first languages whose design was influenced by λ-calculus.

The Church-Rosser theorem [church-rosser] implies the consistency of λ-calculus. The theorem is of great significance in the theory and implementation of programming languages. The history behind the theorem and its proofs is also very interesting since it took many attempts and over thirty years to construct a plausible and widely accepted proof of the Church-Rosser theorem. To paraphrase Rosser [rosser]:

> The original proof of CRT [the Church-Rosser theorem] was fairly long and very complicated. . . . Newman generalized the universe of discourse . . . He proved a result similar to CRT by topological arguments. Curry . . . generalized the Newman result . . . Unfortunately, it turned out that neither the Newman result nor the Curry generalization entailed CRT. . . . This was discovered by Schroer . . . Schroer derived still further generalizations of the Newman and Curry results, which indeed do entail CRT. . . . Schroer 1965 is 627 typed pages . . . Chapter 4 of Curry and Feys 1958 is devoted to a proof of CRT for λ-calculus and . . . is not recommended for light reading. . . . Meanwhile a genuine simplification of the proof of CRT had come in sight. See Martin-Lof 1972. It is agreed that Martin-Lof got some of his ideas from lectures by W. Tait. An exposition of the proof of CRT according to Tait and Martin-Lof appears in Appendix I of Hindley, Lercher and Seldin 1972.

The mechanical proof presented in this paper is based on the Tait/Martin-Lof proof. Since the proof was a relatively recent one and involved considerable combinatorial case-analysis, it made an interesting candidate for mechanical verification. A part of the proof involves the use of a representation of λ-calculus terms due to de Bruijn [debruijn]. The mechanization of this proof also illustrates the efficacy of the Boyer-Moore logic and the accompanying theorem-proving heuristics in stating and proving difficult and deep theorems of a

metamathematical nature.

The Boyer-Moore theorem prover [boyer] is an interactive system for the verification of the properties of pure-Lisp programs. A distinguishing feature of the theorem prover is its powerful use of mathematical induction. The theorem prover has been used to prove many important theorems in number theory, recursive function theory, program verification, etc. It is only recently that the theorem prover has been applied to proofs in the metamathematics of formal systems [metamathematics]. Mechanical proofs in metamathematics have applications in proof-checking and program verification.

The rest of this paper is organized as follows: Section 2 is a brief introduction to λ-calculus. In Section 3, the Church-Rosser theorem is stated and its proof is informally sketched. In both Sections 2 and 3, the standard notation for λ-calculus will be used. Section 4 is a brief introduction to the Boyer-Moore logic in which the mechanical proof was formalized. Section 5 describes the formalization of λ-calculus and a statement of the Church-Rosser theorem in the Boyer-Moore logic. The same formalization in terms of the de Bruijn representation for λ-calculus terms is carried out in Section 6. Section 7 covers the highlights of the mechanical proof. In it, the key lemmas are stated without proof. The machine-generated proofs of some of the key lemmas are presented in Appendix A. In Section 8, conclusions are drawn based on the mechanical proof. Appendix B is a complete list of events (definitions and lemmas) in the proof.

2. An Introduction to Lambda Calculus

This section presents the λ-calculus notions needed to understand the proof. The formalization of λ-calculus within the Boyer-Moore logic will be presented in Section 5. A clear exposition of λ-calculus and the related topic of *combinatory logic* can be found in the book *Introduction to Combinatory Logic* by Hindley, Lercher and Seldin [hindley]. This book also contains a presentation of the proof of the Church-Rosser theorem. Barendregt's *The Lambda Calculus* [barendregt] is a comprehensive volume on the subject.

2.1 Terms

The formal system of Lambda Calculus consists of *terms* (or λ-terms) and certain rules for transforming terms. Intuitively, terms in λ-calculus denote one-argument functions. Terms are constructed starting from *constants* and *variables* (collectively labelled *atoms*). Some confusion might arise from the use of the words 'constant' and 'variable', but unless otherwise mentioned, these refer to λ-calculus constants and variables. Constants do not play any role in the proof. The syntactic variables **a, b, c,** will be used to indicate constants. A term that is a variable represents an arbitrary one-argument function. The syntactic variables **x, y, z,** etc., will be used to represent variables. The actual form of the variables will not be specified except to say that they form a denumerable set and are distinguishable from the constants. The bold-face uppercase letters **A, B, C,** etc., will be used as syntactic variables for terms. There are two ways of forming new terms. The first of these is termed λ-*abstraction*. If **M** is a term and **x** is a variable, then the term **(λx M)** represents the λ-abstraction of **M** with respect to **x**. This means that **(λx M)** is now a one-argument function with a dummy argument **x**. The second way of forming terms is called an *application*. The term **(X Y)** represents result of applying the function **X** to the argument **Y**. The argument to a function and the resulting value can both be functions. The process of *evaluating* a function application will be discussed later.

Thus, the notion of a term in λ-calculus has been defined inductively by showing how terms can be constructed starting from atoms (the base case), by the operations of λ-abstraction and application (the inductive cases). This inductive definition provides a corresponding recursion which will be used extensively. These are definitions where the base case is defined for atoms; the recursive call for a term **(λx M)** is on **M**; and the recursive calls for a term **(M N)** are on the terms **M** and **N**. The phrase, *''recursion on the structure of a term''*,

is used to identify a recursion of this kind. Correspondingly, many proofs involve *induction on the structure of a term*.

Examples of terms:

a,	**(x a)**,	**(λx a)**,
(λx x),	**(λx y)**,	**(λx (a x))**,
((λx (x x)) (λx (x x))),		**(((λx (λy y)) u) v)**,
(λx (λy (λz (x (y z))))),		**(λx (λy (λz ((x z)(y z)))))**.

The expression **X #=# Y** can be read as ‘‘**X** is identical to **Y**’’. In situations where new syntactic variables are introduced in the right-hand side, ‘**#=#**’ is usually read as ‘‘of the form’’, e.g., **X #=# (λx M)**, is read as ‘‘**X** is of the form **(λx M)**’’.

A few relations on terms need to be defined before the transformations on terms can be described. **X** is said to be a *subterm* of **Y** if either **X #=# Y**; or **Y** is of the form **(λx M)** and **X** is a subterm of **M**; or if **Y** is of the form **(M N)** and **X** is a subterm of either **M** or **N**.

Examples:

a is a subterm of **(λx a)**,
(λx x) is a subterm of **(x (λx x))**, and
(λx (x x)) is a subterm of **((λx (x x)) (λx (x x)))**.

The phrase, **X** *occurs in* **Y**, is an alternate way of saying **X** is a subterm of **Y**. An *occurrence* of **X** in **Y** refers to a specific location in **Y** where **X** occurs. The term **(λx (x x))** has two occurrences in the term **((λx (x x)) (λx (x x)))**.

If **(λx M)** is a subterm of a term **Y**, then all the occurrences of **x** in **M** are said to be *bound* in **Y**. Those variable occurrences which are not bound in **Y** are said to be *free* in **Y**. For example, **y** is free but **x** is bound in the term **(λx (x y))**. These notions will now be used to define the important operation of *substitution*.

2.2 Substitution

Substitution is the operation of replacing all the free occurrences of a variable in a term by another term. The result of substituting **X** for the variable **x** in **Y** will be denoted by **[X/x]Y** (to be read as ‘‘**X** for **x** in **Y**’’). To ensure that such a substitution has the intended meaning, no free variable occurrence in **X** can be allowed to become bound in **[X/x]Y**. If this holds of the given **X, x**, and **Y**, then **X** is said to be *free for* **x** in **Y**. Substitution is defined recursively as follows:

[X/x]x #=# X

[X/x]y #=# y, if **x #=#** ~~**#**~~ **y**

[X/x]a #=# a

[X/x](λy M) #=# (λy M), if **x #=# y**, and
#=# (λy [X/x]M), otherwise.

[X/x](M N) #=# ([X/x]M [X/x]N).

In the remainder of this paper, whenever the expression **[X/x]Y** is used, it will be assumed that **X** is free for **x** in **Y**, unless otherwise specified.

Examples: **[a/x](λx x) #=# (λx x)**
[(λx (x x))/x](x x) #=# ((λx (x x)) (λx (x x)))
[(λx y)/x](λy x) #=# (λy (λx y)) (Note: **(λx y)** is not free for **x** in **(λy x)**).

2.3 α-steps and β-steps

There are two rules in λ-calculus for transforming terms. Both the rules will be defined as relations between a term and its transformed version. The first rule permits the renaming of bound variables and is called an α-*step*. **X** goes to **Y** in an α-step (denoted by '**X** $\overset{\#}{\alpha}\rightarrow$**# Y**') iff[1] **X** and **Y** are identical except that the names used to denote the occurrences of variables bound by a λ in **X** and the corresponding λ in **Y** might uniformly differ. This vague definition of an α-step will be made precise in Section 5.

Examples: **((λx x) z)** $\overset{\#}{\alpha}\rightarrow$**# ((λy y) z)**
(λx (x (λy (x y)))) $\overset{\#}{\alpha}\rightarrow$**# (λy (y (λx (y x))))**
((λx (λy (x y))) (λz (y z))) $\overset{\#}{\alpha}\rightarrow$**# ((λx (λz (x z))) (λz (y z)))**

α-steps can be used to rename bound variables in a term **Y** to ensure that a given term **X** is free for **x** in the transformed version of **Y**.

The *evaluation* of terms takes place through a rule known as a β-*step*. It is similar to the operation of replacing dummy parameters with actual parameters in a sub-program. A subterm of the form **((λx M) N)** is called a *redex*. A redex **((λx M) N)** is reduced by means of β-*reduction* to the term that is got by replacing all the free occurrences of **x** in **M** by **N**, i.e., the term **[N/x]M**. A term **X** goes to **Y** in a β-step iff **Y** is got by replacing some non-overlapping redexes in **X** by their β-reduced forms. Two subterms of a term overlap if they both contain the same occurrence of at least one subterm. So, the β-reductions in a β-step must be such that they do not affect one another. The relation **X** goes to **Y** in a β-step, will be denoted by '**X** $\overset{\#}{\beta}\rightarrow$**# Y**'.

Examples: 1. **((λx (a x)) b)** $\overset{\#}{\beta}\rightarrow$**# (a b)**
2. **(((λx y) a) ((λx (x x)) b))** $\overset{\#}{\beta}\rightarrow$**# (y (b b))**
3. **((λx (x x)) (λx (x x)))** $\overset{\#}{\beta}\rightarrow$**# ((λx (x x)) (λx (x x)))**
4. **((λy ((λx x) y)) ((λx (x x)) a))** $\overset{\#}{\beta}\rightarrow$**# ((λx x) ((λx (x x)) a))**
5. **((λy ((λx x) y)) ((λx (x x)) a))** $\overset{\#}{\beta}\rightarrow$**# ((λy y) (a a))**

Note that in the examples 4 and 5 above, the same term is transformed to two different terms by the two β-steps. These two examples will be referred to in the next section.

A term **X** *reduces to* a term **Y** iff **Y** is obtained from **X** by a finite (possibly empty) series of α-steps or β-steps. This means either **X #=# Y** or there must exist terms $\mathbf{Z_0}, \ldots, \mathbf{Z_n}$ such that **X #=#** $\mathbf{Z_0}$, **Y #=#** $\mathbf{Z_n}$, and for all i (0@z< >i<n), either $\mathbf{Z_i} \overset{\#}{\alpha}\rightarrow$**#** $\mathbf{Z_{i+1}}$ or $\mathbf{Z_i} \overset{\#}{\beta}\rightarrow$**#** $\mathbf{Z_{i+1}}$. The relation **X** reduces to **Y** is denoted simply as **X #→# Y**.

In summary, λ-calculus consists of terms and transformations on terms. Terms are either atoms, λ-abstractions or applications. Substitution is the operation of replacing all the occurrences of a certain free variable in one term by another term. There are two rules for transforming terms: α-steps and β-steps. A sequence of α-steps and β-steps can be used to reduce one term to another. These basic λ-calculus notions will be used in the next section to state the Church-Rosser theorem and to sketch a proof of it.

---

[1] *iff* is an abbreviation for *if and only if*.

3. A Proof-sketch of the Church-Rosser Theorem

In this section, the Church-Rosser theorem will be stated and an outline of the proof will be given. The outline of the proof given in this section also applies to the mechanical proof, though the details of the proof differ.

3.1 The Statement

Given the definitions stated in the previous section, the Church-Rosser theorem [church-rosser] can be stated as follows:

If **X** #→# **Y** *and* **X** #→# **Z**, *then there exists a* **W** *such that* **Y** #→# **W** *and* **Z** #→# **W**.

The theorem is depicted pictorially in Figure DIAMOND. It can therefore be said to assert the *Diamond Property* of the relation '#→#'.

Figure 1: The Church-Rosser theorem

The remainder of this section provides a brief outline of the key steps in the proof of the Church-Rosser theorem. The notion of one relation being the *transitive closure* of another relation plays an important role in the proof. A relation (**X** *R* **Y**) is the transitive closure of another relation (**U** *S* **W**) iff the following holds of *R* and *S*:

$\forall$**X, Y** [**X** *R* **Y**
if and only if for some n
$\exists \mathbf{Z_0}, \ldots, \mathbf{Z_n}$ (**X** #=# $\mathbf{Z_0}$ *and*
Y #=# $\mathbf{Z_n}$ *and*
$\forall \mathbf{i}(\mathbf{0} \# \leq \# \mathbf{i} < \mathbf{n})$ $\mathbf{Z_i}$ *S* $\mathbf{Z_{i+1}}$)]

The proof of the Church-Rosser theorem consists of the following steps:

1. Showing that the transitive closure of a relation with the Diamond property has the Diamond property.
2. Defining the relation **X** *walks to* **Y**, such that the relation **X** #→# **Y** is its transitive closure.
3. Proving that the *walks to* relation has the Diamond property.

Clearly, if we can carry out the above three steps, we have a proof of the Diamond property of the relation '#→#', and hence the Church-Rosser theorem.

3.2 The Diamond Property over Transitive Closures

A diagram is employed to establish step 1. Though this diagram is given in terms of the relations **X** *reduces to* **Y**, and **X** *walks to* **Y**, no property of these relations is used in the justification save for the fact that the former is the transitive closure of the latter. The relation **X** *walks to* **Y** is represented by '**X #→# Y**'. Since the relation **X #→# Y** is the transitive closure of **X #→# Y**, it can be represented by a series of walks as **X #=# Y_0 #→# Y_1 #→# . . . #→# Y_{n-1} #→# Y_n #=# Y**. Figure CLOSURE shows how the diamond for relation **X #=# Y** can be constructed from the diamonds for the relation **X #→# Y**. Given that **X #→# Y** and **X #→# Z**, the figure shows how a **W** such that **Y #→# W** and **Z #→# W** can be constructed by repeatedly constructing the smaller diamonds along a row to derive $\mathbf{W_1}$, $\mathbf{W_2}$, and so on.

Figure 2: The Diamond property over transitive closures

3.3 The Definition of Walk

The second step in the proof is to define the notion of a walk or the relation **X #→# Y**. This relation must have the Diamond property and the relation **X #→# Y** must be definable as its transitive closure. The relation **X** *goes to* **Y** *in a single* α *or* β *step* has the second property but does not possess the Diamond property. The appropriate counter-example is given below.

Counter-example:

It can be constructed using only β-steps. Consider the examples 4 and 5 that were used to illustrate β-steps. Let **X** be **((λy ((λx x) y)) ((λx (x x)) a))**, and **Y** be **((λx x) ((λx (x x)) a))**, and let **Z** be **((λy y) (a a))**. As the examples show, **X** $\xrightarrow{\beta}$ **Y** and **X** $\xrightarrow{\beta}$ **Z**. The task is to construct a **W** such that **Y** $\xrightarrow{\beta}$ **W** and **Z** $\xrightarrow{\beta}$ **W**. Clearly, **W** is not one of **Y** or **Z**. The only other possibility is **(a a)**. **Z** goes to **(a a)** in a single β-step, but **Y** needs two overlapping β-reductions. Overlapping β-reductions are not permitted in a β-step.

One point to notice in the above counter-example is that though **Y** could only go to **(a a)** by two overlapping β-reductions, they were such that the inner of these two reductions could be performed first. This example suggests working with a less restricted relation than a β-step in which overlapping β-reductions were permitted, provided the inner redex is reduced before the outer one. Note that when two subterms overlap, one must always be the subterm of the other. Therefore it makes sense to talk of the reduction on an inner redex and an outer one. In any case, this less restricted form of a β-step turns out to work. This new step is called a *walk*. The rest of the proof makes the definition of a walk precise, and demonstrates that it has the Diamond property.

For reasons of brevity, α-steps will be ignored in the remainder of the discussion and it will be assumed that for any redex **((λx M) N)**, **N** is always free for **x** in **M**.

At this point, the reader is urged to make an attempt at defining a walk and showing that it has the Diamond property.

The relation **X** #→# **Y** (**X** walks to **Y**) is defined recursively as follows:

If **X** is an atom, then **X** #→# **Y** iff **Y** #=# **X**.

If **X** #=# **(λx M)**, then **X** #→# **Y** iff **Y** #=# **(λx** $\mathbf{M_Y}$**)** and **M** #→# $\mathbf{M_Y}$.

If **X** #=# **(M N)**, then **X** #→# **Y** iff either:

1. **Y** #=# **(**$\mathbf{M_Y}$ $\mathbf{N_Y}$**)**, **M** #→# $\mathbf{M_Y}$, and **N** #→# $\mathbf{N_Y}$, or
2. **M** #=# **(λx A)**, $\mathbf{M_Y}$ #=# **(λx** $\mathbf{A_Y}$**)**, **M** #→# $\mathbf{M_Y}$, **N** #→# $\mathbf{N_Y}$, and **Y** #=# **[**$\mathbf{N_Y}$**/x]**$\mathbf{A_Y}$].

The above definition employs a recursion on the structure of the term **X**. Clearly, a single β-step can be represented as a walk. It is also the case that any walk can be represented as a series of β-steps. Therefore, the relation **X** *reduces to* **Y** forms the transitive closure of the relation **X** *walks to* **Y**. The above definition of a walk plays an important role in the proof of the Diamond property of walks.

3.4 The Diamond Property of Walks

All that is needed now to complete this sketch of the proof of the Church-Rosser theorem is a proof of the Diamond property of walks. This property could be stated as:

If **X** #→# **Y** and **X** #→# **Z**, then there exists a **W** such that **Y** #→# **W** and **Z** #→# **W**.

The proof is by induction on the structure of the term **X**. Three cases arise and in each case, the appropriate **W** is constructed as follows:

Case 1: [**X** is an atom]

If **X** is an atom, then by the definition of a walk, **X** #=# **Y** #=# **Z**. Then, let **W** be **X** itself. By the definition of a walk, **Y** #→# **W** and **Z** #→# **W**.

Case 2: [**X** #=# **(λx M)**]

By the definition of a walk, there must exist $\mathbf{M_Y}$ and $\mathbf{M_Z}$ such that **Y** #=# **(λx** $\mathbf{M_Y}$**)** and **Z** #=# **(λx** $\mathbf{M_Z}$**)**, and

M #→# $\mathbf{M_Y}$ (1)
M #→# $\mathbf{M_Z}$. (2)

Applying the Induction Hypothesis to **M** given (1) and (2), yields $\mathbf{M_W}$ such that $\mathbf{M_Y}$ #→# $\mathbf{M_W}$ and $\mathbf{M_Z}$ #→# $\mathbf{M_W}$. Let **W** be **(λx** $\mathbf{M_W}$**)**. Then, by the definition of a walk, both **Y** and **Z** walk to **W**.

Case 3: [**X** #=# **(M N)**]

This splits up into 5 subcases depending on whether or not **X** is a redex, and if it is, whether or not that redex is β-reduced in the walk. The subcases are as follows:

1. **X** is not a redex.
2. **X** is a redex and is only β-reduced in the walk to **Y**.
3. **X** is a redex and is only β-reduced in the walk to **Z**.
4. **X** is a redex and is β-reduced in both the walk to **Y** and to **Z**.
5. **X** is a redex but is not β-reduced in the walk to **Y** or to **Z**.

The subcases 1 and 5 turn out to have the same proof. Subcases 2 and 3 are symmetrical and so the proof of subcase 3 will be omitted.

Subcases 1, 5: [X is not a redex or is a redex which is not β-reduced in the walks to **Y** and **Z**.]

By the definition of a walk, there must exist $\mathbf{M_Y}$, $\mathbf{N_Y}$, $\mathbf{M_Z}$, $\mathbf{N_Z}$ such that $\mathbf{Y \#=\# (M_Y\ N_Y)}$ and $\mathbf{Z \#=\# (M_Z\ N_Z)}$, and

$$\mathbf{M \#\rightarrow\# M_Y;} \quad \mathbf{N \#\rightarrow\# N_Y} \quad (3)$$
$$\mathbf{M \#\rightarrow\# M_Z;} \quad \mathbf{N \#\rightarrow\# N_Z}. \quad (4)$$

Applying the Induction Hypothesis to both **M** and **N** given (3) and (4), we derive $\mathbf{M_W}$ and $\mathbf{N_W}$ such that

$$\mathbf{M_Y \#\rightarrow\# M_W;} \quad \mathbf{M_Z \#\rightarrow\# M_W}$$

(5)

$$\mathbf{N_Y \#\rightarrow\# N_W;} \quad \mathbf{N_Z \#\rightarrow\# N_W.} \quad (6)$$

Now if we let **W** be $\mathbf{(M_W\ N_W)}$, we can conclude from (5), (6) and the definition of a walk that both **Y** and **Z** walk to **W**.

The remaining subcases employ a lemma called the *Substitutivity of Walks* which asserts that if $\mathbf{A_Y}$ and $\mathbf{N_Y}$ walk to $\mathbf{A_W}$ and $\mathbf{N_W}$ respectively, then the result of substituting $\mathbf{N_Y}$ for **x** in $\mathbf{A_Y}$, walks to the result of substituting $\mathbf{N_W}$ for **x** in $\mathbf{A_W}$. This will be stated here without proof as:

Lemma: *If* $\mathbf{A_Y \#\rightarrow\# A_W}$ *and* $\mathbf{N_Y \#\rightarrow\#\ \ N_W}$, *then* $\mathbf{[N_Y/x]A_Y \#\rightarrow\# [N_W/x]A_W}$.

Subcase 2: [**X** is a redex which is only reduced in the walk to **Y**]

Since **X** is a redex, let $\mathbf{X \#=\# ((\lambda x\ A)\ N)}$. From the definition of a walk applied to $\mathbf{X \#\rightarrow\# Y}$ and $\mathbf{X \#\rightarrow\# Z}$, there must exist $\mathbf{A_Y}$, $\mathbf{N_Y}$, $\mathbf{A_Z}$, $\mathbf{N_Z}$ such that $\mathbf{Y \#=\# [N_Y/x]A_Y}$ and $\mathbf{Z \#=\# ((\lambda x\ A_Z)\ N_Z)}$, and

$$\mathbf{A \#\rightarrow\# A_Y;} \quad \mathbf{N \#\rightarrow\# N_Y} \quad (7)$$
$$\mathbf{A \#\rightarrow\# A_Z;} \quad \mathbf{N \#\rightarrow\# N_Z}. \quad (8)$$

Therefore, by the Induction Hypothesis applied to **A** and **N** and given (7) and (8), there must exist $\mathbf{A_W}$ and $\mathbf{N_W}$ such that

$$\mathbf{A_Y \#\rightarrow\# A_W;} \quad \mathbf{A_Z \#\rightarrow\# A_W} \quad (9)$$
$$\mathbf{N_Y \#\rightarrow\# N_W;} \quad \mathbf{N_Z \#\rightarrow\# N_W}. \quad (10)$$

Let **W** be $\mathbf{[N_W/x]A_W}$. It can be shown that **Y** (of the form $\mathbf{[N_Y/x]A_Y}$) walks to **W** (of the form $\mathbf{[N_W/x]A_W}$), from (9), (10) and *Substitutivity of Walks* lemma. Showing that **Z** (of the form $\mathbf{((\lambda x\ A_Z)\ N_Z)}$) walks to **W** involves (9), (10) and the definition of a walk.

Subcase 4: [**X** is a redex and is reduced in the walks to both **Y** and **Z**]

This subcase is very similar to the previous one. As before, let $\mathbf{X \#=\# ((\lambda x\ A)\ N)}$. The definition of a walk yields (7) and (8) as before but with $\mathbf{Z \#=\# [N_Z/x]A_Z}$. The same Induction Hypothesis applied to **A** and **N** leads to (9) and (10) respectively. The argument to show that $\mathbf{Y \#\rightarrow\# W}$ is the same as in the previous subcase. Showing that **Z** (of the form $\mathbf{[N_Z/x]A_Z}$) walks to **W** (of the form $\mathbf{[N_W/x]A_W}$) employs (9), (10) and the *Substitutivity of Walks* lemma.

This completes the proof of the Diamond property of walks. The proof of the *Substitutivity of Walks* lemma proceeds by a similar induction on the structure of the term $\mathbf{A_Y}$. This proof requires the property of substitution which permits us to exchange the order of two successive substitutions. The lemma below asserts that the result

of substituting a term **N** for **x** in **[Z/y]M**, is the same as the result of substituting **[N/x]Z** for **y** in **[N/x]M**, where **x** is different from **y**.

Lemma: [N/x][Z/y]M @eqer for the shell is the 1-place function NLAMBDAP**, and** (NLAMBDAP (NLAMBDA X Y)) **returns** T**.**

Applications are represented by another shell in which the constructor is the 2-place function NCOMB**.**

Examples of terms:

	Term	Formal Representation
1.	**((λx (x x))(λx (x x)))**	(NCOMB (NLAMBDA 1 (NCOMB 1 1)) (NLAMBDA 1 (NCOMB 1 1)))
2.	**(λx (a x))**	(NLAMBDA 0 (NCOMB 'A 0))
3.	**(λx (λy (λz ((x z)(y z)))))**	(NLAMBDA 1 (NLAMBDA 2 (NLAMBDA 3 (NCOMB (NCOMB 1 3) (NCOMB 2 3)))))

Having described the manner in which terms are represented, we can now examine the function that ch

```
9.   Definition.
     (NTERMP X)
        =
     (IF (NLAMBDAP X)
         (NTERMP (NBODY X))
         (IF (NCOMBP X)
             (AND (NTERMP (NLEFT X))
                  (NTERMP (NRIGHT X)))
             (OR (NUMBERP X) (LITATOM X))))
```

The next step is to define the operation of substitution. The function NSUBST **takes three arguments** X**,**

```
4.   Definition.
     (NSUBST X Y N)
        =
     (IF (NLAMBDAP X)
         (IF (EQUAL (NBIND X) N)
             X
             (NLAMBDA (NBIND X)
                      (NSUBST (NBODY X) Y N)))
         (IF (NCOMBP X)
             (NCOMB (NSUBST (NLEFT X) Y N)
                    (NSUBST (NRIGHT X) Y N))
             (IF (NUMBERP X)
                 (IF (EQUAL X N) Y X)
                 X)))
```

Examples:

1. **[a/x](λx x)**
   @z< >
   (λx x)

```
(NSUBST (NLAMBDA 1 1) 'A 1)
=
(NLAMBDA 1 1)
```

2. **[(λx (x x))/x](x x)**
   @z<

```
(NSUBST (NCOMB 1 1)
        (NLAMBDA 1 (NCOMB 1 1)) 1)
```

=

((λx (x x))(λx (x x)))

```
(NCOMB (NLAMBDA 1 (NCOMB 1 1))

(NLAMBDA 1 (NCOMB 1 1)))
```

3.
[(λx y)/x](λy x)

```
(NSUBST (NLAMBDA 2 1) (NLAMBDA 1 2) 1)
```

@z< >
=

(λy (λx y))

```
(NLAMBDA 2 (NLAMBDA 1 2))
```

(Note: the free variable y is captured in this substitution.)

>

The function NOT-FREE-IN **used as** (NOT-FREE-IN X Y) **checks if the variable** X **does not occur free**

The next definition provides a formal definition of an α-step. The function ALPHA-EQUAL **checks if tw**

```
7.   Definition.
     (INDEX N LIST)
        =
     (IF (LISTP LIST)
         (IF (EQUAL (CAR LIST) N)
             1
             (ADD1 (INDEX N (CDR LIST))))
         (ADD1 N))
```

```
8.   Definition.
     (ALPHA-EQUAL A B X Y)
        =
     (IF (AND (NLAMBDAP A) (NLAMBDAP B))
         (ALPHA-EQUAL (NBODY A)
                      (NBODY B)
                      (CONS (NBIND A) X)
                      (CONS (NBIND B) Y))
         (IF (AND (NCOMBP A) (NCOMBP B))
             (AND (ALPHA-EQUAL (NLEFT A) (NLEFT B) X Y)
                  (ALPHA-EQUAL (NRIGHT A)
                               (NRIGHT B)
                               X Y))
             (IF (AND (NUMBERP A) (NUMBERP B))
                 (EQUAL (INDEX A X) (INDEX B Y))
                 (EQUAL A B))))
```

Examples:

1. **((λx x) z)** $\overset{\#}{\underset{\alpha}{\rightarrow}}\#$ **((λy y) z)**

```
(ALPHA-EQUAL (NCOMB (NLAMBDA 1 1) 3)
             (NCOMB (NLAMBDA 2 2) 3)  NIL NIL)
```

2. **(λx (x (λy (x y))))** $\overset{\#}{\underset{\alpha}{\rightarrow}}\#$ **(λy (y (λx (y x))))**

```
(ALPHA-EQUAL (NLAMBDA 1 (NCOMB 1 (NLAMBDA 2 (NCOMB 1 2))))
             (NLAMBDA 2 (NCOMB 2 (NLAMBDA 1 (NCOMB 2 1))))
             NIL NIL)
```

3. **((λx (λy (x y))) (λz (y z)))** $\overset{\#}{\underset{\alpha}{\rightarrow}}\#$ **((λx (λz (x z))) (λz (y z)))**

```
(ALPHA-EQUAL (NCOMB (NLAMBDA 1 (NLAMBDA 2 (NCOMB 1 2)))
                    (NLAMBDA 3 (NCOMB 2 3)))
             (NCOMB (NLAMBDA 1 (NLAMBDA 3 (NCOMB 1 3)))
                    (NLAMBDA 3 (NCOMB 2 3)))
             NIL NIL)
```

The next important definition captures the relation A $\overset{\#}{\underset{\beta}{\rightarrow}}\#$ **B. The function** NBETA-STEP **takes two arg**

```
10.  Definition.
     (NBETA-STEP A B)
        =
     (IF (EQUAL A B)
         T
         (IF (NLAMBDAP A)
             (AND (NLAMBDAP B)
                  (EQUAL (NBIND A) (NBIND B))
                  (NBETA-STEP (NBODY A) (NBODY B)))
             (IF (NCOMBP A)
                 (OR (AND (NLAMBDAP (NLEFT A))
                          (FREE-FOR (NBODY (NLEFT A))
                                    (NRIGHT A))
                          (EQUAL B
                                 (NSUBST (NBODY (NLEFT A))
                                         (NRIGHT A)
                                         (NBIND (NLEFT A)))))
                     (AND (NCOMBP B)
                          (NBETA-STEP (NLEFT A) (NLEFT B))
                          (NBETA-STEP (NRIGHT A) (NRIGHT B))))
                 F)))
```

Examples:

1. **((λx (a x)) b) $\#_{\beta}\!\!\rightarrow\!\#$ (a b)**

```
(NBETA-STEP (NCOMB (NLAMBDA 1 (NCOMB 'A 1)) 'B)
            (NCOMB 'A 'B))
```

2. **((λy ((λx x) y)) ((λx (x x)) a)) $\#_{\beta}\!\!\rightarrow\!\#$ ((λx x)((λx (x x)) a))**

```
(NBETA-STEP (NCOMB (NLAMBDA 2 (NCOMB (NLAMBDA 1 1) 2))
                   (NCOMB (NLAMBDA 1 (NCOMB 1 1)) 'A))
            (NCOMB (NLAMBDA 1 1)(NCOMB (NLAMBDA 1 (NCOMB 1 1)) 'A)))
```

3. **((λy ((λx x) y)) ((λx (x x)) a)) $\#_{\beta}\!\!\rightarrow\!\#$ ((λy y) (a a))**

```
(NBETA-STEP (NCOMB (NLAMBDA 2 (NCOMB (NLAMBDA 1 1) 2))
                   (NCOMB (NLAMBDA 1 (NCOMB 1 1)) 'A))
            (NCOMB (NLAMBDA 2 2) (NCOMB 'A 'A)))
```

The last definition is that of a reduction and is quite straightforward. The function `NREDUCTION` **chec**

```
11.  Definition.
     (NSTEP A B)
        =
     (OR (ALPHA-EQUAL A B NIL NIL)
         (NBETA-STEP A B))
```

```
12.  Definition.
     (NREDUCTION A B LIST)
        =
     (IF (LISTP LIST)
         (AND (NSTEP (CAR LIST) B)
              (NREDUCTION A (CAR LIST) (CDR LIST)))
         (NSTEP A B))
```

Finally, the statement of the Church-Rosser theorem expressed in terms of this formalization would read

```
REDS-Y,  REDS-Z,  W  (IMPLIES (AND (NTERMP X)
                                   (NREDUCTION X Y LIST1)
                                   (NREDUCTION X Z LIST2))
                              (AND (NREDUCTION Y W REDS-Y)
                                   (NREDUCTION Z W REDS-Z)))
```

In English, if `X` **reduces to** `Y` **via** `LIST1`**, and to** `Z` **via** `LIST2`**, then there exist** `REDS-Y, REDS-Z,` **and**

```
157. Theorem.  FINALLY-CHURCH-ROSSER (rewrite):
     (IMPLIES (AND (NTERMP X)
                   (NREDUCTION X Y LIST1)
                   (NREDUCTION X Z LIST2))
              (AND (NREDUCTION Y
                               (MAKE-N-W X Z Y LIST2 LIST1)
                               (NMAKE-REDUCTION X Y Z LIST1 LIST2))
                   (NREDUCTION Z
                               (MAKE-N-W X Z Y LIST2 LIST1)
                               (NMAKE-REDUCTION X Z Y LIST2 LIST1))))
```

Thus, the λ-calculus described in Section 2 has been formalized in the Boyer-Moore logic. The statement

4. The Formalization in the de Bruijn Notation

In this section, the Lambda Calculus will be formally defined using a notation due to de Bruijn and the

1. **Discussion of the de Bruijn notation for λ-terms.**
2. **Definition of β-reduction.**
3. **Definition of a walk.**
4. **Statement of the Diamond property for walks.**
5. **Statement of the Church-Rosser theorem.**

4.1 The de Bruijn Notation

As mentioned earlier, a part of the mechanical proof employs a notation for λ-calculus terms that is diffe

The de Bruijn notation is drastically different from the standard notation used in Sections 2, 3 and 5. A f

The de Bruijn notation used in this proof does away with names for variables and tags for λ while retaini

One important difference between the standard notation and the de Bruijn notation is that in the de Bru

Another important difference is that since variables do not have names in the de Bruijn notation, it make

The representation of these terms in the Boyer-Moore logic is quite similar to that described in the previ

The second shell consists of a 1-place recognizer `COMBP`**; a 2-place constructor** `COMB`**; and two 1-place**

Variables are represented by the positive natural numbers 1, 2, 3, etc. The literal atoms in the Boyer-Mo

Example:

Standard Notation	**de Bruijn notation**
(λx x)	`(LAMBDA 1)`
(λx (λy (λz (x (y z)))))	`(LAMBDA (LAMBDA` `(LAMBDA (COMB 3 (COMB 2 1)))))`

(λx ((λy (x (y z))) (x z)))	`(LAMBDA (COMB`
	`(LAMBDA (COMB 2 (COMB 1 3)))`
	`(COMB 1 2)))`

We can now examine the definition of the function which translates a term in the standard notation to th

```
50.  Definition.
     (TRANSLATE X BOUNDS)
        =
     (IF (NLAMBDAP X)
         (LAMBDA (TRANSLATE (NBODY X)
                            (CONS (NBIND X) BOUNDS)))
         (IF (NCOMBP X)
             (COMB (TRANSLATE (NLEFT X) BOUNDS)
                   (TRANSLATE (NRIGHT X) BOUNDS))
             (IF (NUMBERP X) (INDEX X BOUNDS) X)))
```

To get a better grasp of the de Bruijn notation, let us write a few programs in pidgin pure-Lisp that man

BEEP **works by recursing down the structure of the term** X **incrementing its counter by one, each time it**

```
(DEFN BEEP (X N)
  (IF (LAMBDAP X)
      (BEEP (BODY X) (ADD1 N))
      (IF (COMBP X)
          (AND (BEEP (LEFT X) N)
               (BEEP (RIGHT X) N))
          (IF (LESSP N X)
              "beep"
              T))))
```

Let us examine one other program which locates the occurrences of variables in Y **that are bound by the**

```
(DEFN HOOT (X N)
  (IF (LAMBDAP X)
      (HOOT (BODY X) (ADD1 N))
      (IF (COMBP X)
          (AND (HOOT (LEFT X) N)
               (HOOT (RIGHT X) N))
          (IF (EQUAL X N)
              "hoot"
              T))))
```

Thus (HOOT X 1) **locates occurrences of the lowest free variable in** X**, i.e., the one that would be bound**

Both BEEP **and** HOOT **recurse on the structure of terms and will be used as paradigms in explaining so**

4.2 β-Reduction

The definition of β-reduction in this notation is substantially more complicated than the corresponding d

As before, a subterm of the form (COMB (LAMBDA X) Y) **is labelled a** *redex***. A β-reduction should**

The reason we need an appropriate transformation of Y **is to ensure that the free variables in** Y **do not b**

The function BUMP **displayed below is very similar to the previously introduced function** BEEP**. It take**

```
15.  Definition.
     (BUMP X N)
        =
     (IF (LAMBDAP X)
         (LAMBDA (BUMP (BODY X) (ADD1 N)))
         (IF (COMBP X)
             (COMB (BUMP (LEFT X) N)
                   (BUMP (RIGHT X) N))
             (IF (LESSP N X) (ADD1 X) X)))
```

Examples of Bump:

```
   (BUMP (LAMBDA (LAMBDA (COMB 2 3))) 0)  =    (LAMBDA (LAMBDA (COMB 2 4)))

(BUMP (LAMBDA (LAMBDA (COMB 1 2))) 0)  =    (LAMBDA (LAMBDA (COMB 1 2)))

(BUMP (LAMBDA (COMB (LAMBDA 3)(COMB 3 1))) 1)  =    (LAMBDA (COMB (LAMB
```

Now we can use the function BUMP **to define** β**-reduction.** β**-reduction will be defined by means of a func**

The free variables in the (LAMBDA X) **part of the redex are affected because the** LAMBDA **in** (LAM

The function SUBST **combines features of the previously defined functions** BEEP **and** HOOT**. It takes t**

```
16.  Definition.
     (SUBST X Y N)
        =
     (IF (LAMBDAP X)
         (LAMBDA (SUBST (BODY X) (BUMP Y 0) (ADD1 N)))
         (IF (COMBP X)
             (COMB (SUBST (LEFT X) Y N)
                   (SUBST (RIGHT X) Y N))
             (IF (NOT (ZEROP X))
                 (IF (EQUAL X N)
                     Y
                     (IF (LESSP N X) (SUB1 X) X))
                 X)))
```

In the base case, X **is either a constant or a variable. The test** (NOT (ZEROP X)) **is** T **only if** X **is a posi**

If X **is of the form** (LAMBDA M)**, then** SUBST **returns** (LAMBDA M1)**, where** M1 **is the result of the r**

The case when X **is of the form** (COMB U V) **turns out to be quite simple.** SUBST **recurses on** U**, i.e.,** (

Examples of Subst:

```
1.      (SUBST 1 (LAMBDA 1) 1)  =  (LAMBDA 1)

2.      (SUBST (LAMBDA (COMB 2 3))  (LAMBDA (COMB 1 2)) 1)
        =
        (LAMBDA (COMB (LAMBDA (COMB 1 3)) 2))

3.      (SUBST (COMB (LAMBDA (LAMBDA (COMB 1 (COMB 4 5))))
                     (COMB (LAMBDA 3)(LAMBDA 4)))
               1  2)
        =
        (COMB (LAMBDA (LAMBDA (COMB 1 (COMB 3 4))))
              (COMB (LAMBDA 2)(LAMBDA 3)))
```

(SUBST X Y 1) **denotes the result of applying a** β**-reduction to the redex** (COMB (LAMBDA X) Y).

```
70.  Theorem.  TRANSLATE-PRESERVES-REDUCTION (rewrite):
     (IMPLIES (AND (NLAMBDAP X)
                   (FREE-FOR (NBODY X) Y)
                   (NTERMP X)
                   (NTERMP Y))
              (EQUAL (TRANSLATE (NSUBST (NBODY X) Y (NBIND X))
                                BOUNDS)
                     (SUBST (BODY (TRANSLATE X BOUNDS))
                            (TRANSLATE Y BOUNDS)
                            1)))
```

This completes the description of β**-reduction. We can now use this to describe the notion of a walk.**

4.3 Definition of a Walk

In the informal proof-sketch, a walk was described as a sequence of β**-reductions on the redexes of a ter**

The function WALK **below applies a walk-instruction** W **to a term** M **to return the conclusion of the w**

```
29.  Definition.
     (WALK W M)
        =
     (IF (LAMBDAP M)
         (LAMBDA (WALK W (BODY M)))
         (IF (COMBP M)
             (IF (AND (EQUAL (COMMAND W) 'REDUCE)
                      (LAMBDAP (LEFT M)))
                 (SUBST (BODY (WALK (LEFT-INSTRS W) (LEFT M)))
                        (WALK (RIGHT-INSTRS W) (RIGHT M))
                        1)
                 (COMB (WALK (LEFT-INSTRS W) (LEFT M))
                       (WALK (RIGHT-INSTRS W) (RIGHT M))))
             M))
```

The function WALK **is defined by a recursion on the structure of the term** M. **In the base case when** M

1. M **is a redex and** W **contains a** 'REDUCE **in the command position, i.e.,** (COMMAND W).
2. M **is a redex but** 'REDUCE **does not occur in the command position of** W.
3. M **is not a redex.**

Subcases 2 and 3 are dealt with identically. To determine if subcase 1 applies, two tests are made. The

The above notion of a walk can now be used to formally state the Diamond property for walks in the Boy

4.4 The Statement of the Diamond Property for Walks

The statement of the Diamond property from Section 2 was as follows:

If **X** #→# **Y** and **X** #→# **Z**, then there exists a **W** such that **Y** #→# **W** and **Z** #→# **W**.

The task here is to restate the Diamond property using the function WALK **instead of the relation X** #→#

```
W1  W2 (EQUAL (WALK W1 (WALK U M))
              (WALK W2 (WALK V M))).
```

Since the Boyer-Moore logic does not permit the use of quantifiers, the existential quantifiers over W1 **an**

```
41.  Theorem.  MAIN (rewrite):
     (EQUAL (WALK (MAKE-WALK M U V) (WALK U M))
            (WALK (MAKE-WALK M V U) (WALK V M)))
```

The definition of the function MAKE-WALK **will be discussed in the next section. The statement of the**

4.5 The Statement of the Church-Rosser Theorem

At this point, two assumptions will be made. One, that any single β**-step can be represented as a walk.**

```
42.  Definition.
     (REDUCE W M)
        =
     (IF (LISTP W)
         (REDUCE (CDR W) (WALK (CAR W) M))
         M)
```

Given the assumptions made in the previous paragraph, the statement of the Church-Rosser theorem c

```
W1, W2 (EQUAL (REDUCE W1 (REDUCE V M))
              (REDUCE W2 (REDUCE U M))).
```

To relate the above statement to the one given in the Section 3, M **corresponds to X,** (REDUCE U M) **re**

As before, the existential quantifiers over W1 **and** W2 **will have to be replaced by functions. The term**

```
49.  Theorem.  CHURCH-ROSSER (rewrite):
     (EQUAL (REDUCE (MAKE-REDUCE M U V)
                    (REDUCE V M))
            (REDUCE (MAKE-REDUCE M V U)
                    (REDUCE U M)))
```

This concludes the presentation of the formalization of the λ**-calculus using the de Bruijn notation, within**

5. Highlights of the Mechanical Proof

Having formally defined the λ-calculus and stated a version of the Church-Rosser theorem for it, we can

1. **Some lemmas on** SUBST, WALK, **and** BUMP**.**
2. **The Substitutivity of Walks lemma.**
3. **The Diamond property for walks.**
4. **The Church-Rosser theorem**

5.1 Some Properties of BUMP, SUBST, and WALK

The lemmas discussed below state some interesting properties of the functions BUMP, SUBST, **and** WA

The first of these is a straightforward lemma about commuting the order of two successive applications o

```
17.  Theorem.  BUMP-BUMP (rewrite):
     (IMPLIES (LEQ N M)
              (EQUAL (BUMP (BUMP Y N) (ADD1 M))
                     (BUMP (BUMP Y M) N)))
```

The next couple of lemmas will relate the operations BUMP **and** SUBST**. Both demonstrate how** BUMP

```
19.  Theorem.  BUMP-SUBST (rewrite):
     (IMPLIES (LESSP M N)
              (EQUAL (BUMP (SUBST X Y N) M)
                     (SUBST (BUMP X M)
                            (BUMP Y M)
                            (ADD1 N))))
     Hint:  Induct as for (BUMP-SUBST-INDUCT X Y N M).
```

The other lemma relating BUMP **and** SUBST**,** ANOTHER-BUMP-SUBST **deals with the case when the**

```
23.  Theorem.  ANOTHER-BUMP-SUBST (rewrite):
     (IMPLIES (LEQ N (ADD1 M))
              (EQUAL (SUBST (BUMP X (ADD1 M)) (BUMP Y M) N)
                     (BUMP (SUBST X Y N) M)))
     Hint:  Induct as for (BUMP-SUBST-INDUCT X Y N M).
```

There is one other lemma about BUMP **and** SUBST **which is displayed below as** SUBST-NOT-FREE-IN**.**

```
36.  Theorem.  SUBST-NOT-FREE-IN (rewrite):
     (EQUAL (SUBST (BUMP X N) Y (ADD1 N))
            X)
     Hint:  Induct as for (SUBST X Y N).
```

The next lemma is perhaps the most significant one in the proof. Just as BUMP-BUMP **stated that the or**

```
37.  Theorem.  SUBST-SUBST (rewrite):
     (IMPLIES (AND (NUMBERP M) (LESSP M N))
              (EQUAL (SUBST (SUBST X (BUMP Z M) (ADD1 N))
                            (SUBST Y Z N)
                            (ADD1 M))
                     (SUBST (SUBST X Y (ADD1 M)) Z N)))
     Hint:  Induct as for (SUBST-INDUCT X Y Z M N).
```

At first sight, the statement of the above lemma might seem contrived and motivated by some specific re

5.2 The Substitutivity of Walks Lemma

The informal statement of the substitutivity of walks lemma was given in Section 3. It went as follows:

If **M** walks to **M'**, and **N** walks to **N'**, then **[N/x]M** walks to **[N'/x]M'**.

The same lemma stated below will read differently because it is stated in terms of walk-instructions rath

If W and U are two walk-instructions such that W applied to M yields M′, and U applied to X yields X′, then there exists a walk-instruction V such that V applied to (SUBST M X N) yields (SUBST M′ X′ N).

In the statement below, the required walk-instruction V **is constructed by the function** SUB-WALK**, and**

```
38.  Theorem.  WALK-SUBST (rewrite):
     (IMPLIES (NOT (ZEROP N))
              (EQUAL (WALK (SUB-WALK W M U N)
                           (SUBST M X N))
                     (SUBST (WALK W M) (WALK U X) N)))
     Hint:  Induct as for (WALK-SUBST-IND W M U X N).
```

As was seen in the proof-sketch in Section 3, the substitutivity of walks lemma is the key to the proof of t

5.3 The Diamond Property of Walks

The Diamond property of walks was formally stated in the previous section. A function MAKE-WALK

```
40.  Definition.
     (MAKE-WALK M U V)
        =
     (IF (LAMBDAP M)
         (MAKE-WALK (BODY M) U V)
         (IF (COMBP M)
             {code for COMBP case}
             U))
```

In the base case, when M **is neither a** LAMBDAP **nor a** COMBP**,** MAKE-WALK **returns** U**. This is actu**

When M **is of the form** (LAMBDA X)**, where** X **is** (BODY M)**,** MAKE-WALK **recurses on** X**. From th**

In the case when M **is of the form** (COMB X Y)**, where** X **is** (LEFT M) **and** Y **is** (RIGHT M)**, there ar**

```
(IF (AND (EQUAL (COMMAND U) 'REDUCE)
         (LAMBDAP (LEFT M)))
    {code for case when X is a redex and
     U instructs a reduction}
    (IF (AND (EQUAL (COMMAND V) 'REDUCE)
             (LAMBDAP (LEFT M)))
        {code for case when X is a redex and
         V instructs a reduction but U does not}

        {code for case when either X is not a redex
         or neither U nor V instructs a reduction}
         ))
```

If M **is a redex and the walk-instruction command is** 'REDUCE, **then regardless of** V**, the resulting walk-**
X1 **is** (WALK (LEFT-INSTRS U) X),
Y1 **is** (WALK (RIGHT-INSTRS U) Y),
X2 **is** (WALK (MAKE-WALK X (LEFT-INSTRS U)(LEFT-INSTRS V)) X1), **and**
Y2 **is** (WALK (MAKE-WALK Y (RIGHT-INSTRS U)(RIGHT-INSTRS V) Y1).

The code for the above case is given below as:

```
(SUB-WALK (MAKE-WALK (LEFT M)
                     (LEFT-INSTRS U)
                     (LEFT-INSTRS V))
          (BODY (WALK (LEFT-INSTRS U) (LEFT M)))
          (MAKE-WALK (RIGHT M)
                     (RIGHT-INSTRS U)
                     (RIGHT-INSTRS V))
          1)
```

If M **is a redex and walk-instruction** U **does not command a reduction but walk-instruction** V **does, then**

```
(LIST (MAKE-WALK (LEFT M)
                 (LEFT-INSTRS U)
                 (LEFT-INSTRS V))
      (MAKE-WALK (RIGHT M)
                 (RIGHT-INSTRS U)
                 (RIGHT-INSTRS V))
      'REDUCE)
```

The final case is when either M **is not a redex or neither walk-instruction commands a reduction, then** M

```
(LIST (MAKE-WALK (LEFT M)
                 (LEFT-INSTRS U)
                 (LEFT-INSTRS V))
      (MAKE-WALK (RIGHT M)
                 (RIGHT-INSTRS U)
                 (RIGHT-INSTRS V)))))
```

The Diamond construction for the Church-Rosser theorem is much simpler than the above Diamond con

5.4 The Church-Rosser theorem

The statement of the theorem has already been presented in Section 6. The function MAKE-REDUCE **i**

MAKE-REDUCE **takes three arguments: a term** M **and two lists of walk-instructions** U **and** V**. In defi**

```
46.  Definition.
     (MAKE-REDUCE M U V)
        =
     (IF (LISTP U)
         (CONS (MAKE-WALK-REDUCE M (CAR U) V)
               (MAKE-REDUCE (WALK (CAR U) M)
                            (CDR U)
                            (MAKE-REDUCE-WALK M (CAR U) V)))
         U)
```

The key lemma relating the two functions MAKE-WALK-REDUCE **and** MAKE-REDUCE-WALK **is:**

```
45.  Theorem.  WALK-REDUCE (rewrite):
     (EQUAL (REDUCE (MAKE-REDUCE-WALK M W V)
                    (WALK W M))
            (WALK (MAKE-WALK-REDUCE M W V)
                  (REDUCE V M)))
```

The Figure RECTANGLE shows how these functions can be used to build a rectangle consisting of a wal

Figure 3: MAKE-WALK-REDUCE and MAKE-REDUCE-WALK.

Figure MAKE-REDUCE illustrates how MAKE-REDUCE **employs the rectangles built by** MAKE-WAL

Figure 4: MAKE-REDUCE and the Church-Rosser theorem

In stating the Church-Rosser theorem as shown in CHURCH-ROSSER **some assumptions were made. T**

```
74.  Theorem.  MAKE-WALK-STEP-WALKS (rewrite):
     (IMPLIES (BETA-STEP A B)
              (EQUAL (WALK (MAKE-WALK-STEP A B) A)
                     B))

94.  Theorem.  MAKE-REDUCE-STEPS-STEPS (rewrite):
     (REDUCTION A
                (REDUCE W A)
                (MAKE-REDUCE-STEPS W A))
```

Finally, the Church-Rosser theorem was proved in the acceptable form by the two lemmas below, which

```
97.  Theorem.  THE-REAL-CHURCH-ROSSER (rewrite):
     (IMPLIES (AND (REDUCTION X Y LIST1)
                   (REDUCTION X Z LIST2))
              (AND (REDUCTION Y
                              (MAKE-W X Z Y LIST2 LIST1)
                              (MAKE-REDUCTION X Y Z LIST1 LIST2))
                   (REDUCTION Z
                              (MAKE-W X Y Z LIST1 LIST2)
                              (MAKE-REDUCTION X Z Y LIST2 LIST1))))

98.  Theorem.  BOTH-MAKE-W-ARE-SAME (rewrite):
     (IMPLIES (AND (REDUCTION X Y LIST1)
                   (REDUCTION X Z LIST2))
              (EQUAL (MAKE-W X Y Z LIST1 LIST2)
                     (MAKE-W X Z Y LIST2 LIST1)))
```

The theorems THE-REAL-CHURCH-ROSSER **and** BOTH-MAKE-W-ARE-SAME **were used to prove t**

The translation back into the standard notation was done using the function UNTRANS**. The definition**

```
107. Definition.
     (UNTRANS X M N)
        =
     (IF (LAMBDAP X)
         (NLAMBDA (PLUS M (ADD1 N))
                  (UNTRANS (BODY X) M (ADD1 N)))
         (IF (COMBP X)
             (NCOMB (UNTRANS (LEFT X) M N)
                    (UNTRANS (RIGHT X) M N))
             (IF (LESSP N X)
                 (DIFFERENCE X (ADD1 N))
                 (IF (NOT (ZEROP X))
                     (ADD1 (PLUS M (DIFFERENCE N X)))
                     X))))
```

In summary, the lemmas leading to a proof of the Church-Rosser theorem were stated and proved using

6. Conclusions

In the preceding sections, we have discussed the formalization and mechanical proof of the Church-Ross

6.1 Background notes

The Church-Rosser theorem is an extremely important theorem in Lambda Calculus which lacked a wid

In 1983, Boyer and Moore suggested that the proof would make an interesting candidate to try out on th

The results were presented at a weekly seminar conducted by Boyer and Moore. It was encouraging to n

The proof was carried out on a Symbolics 3600 Lisp machine. "Alonzo", "Barkley", and "Haskell" were

6.2 Observations

The above mechanical proof of the Church-Rosser theorem raises several issues that require comment.

The first observation deals with the implications of the success of the mechanical proof attempt. The co

The next observation concerns the level at which the proof was checked. Clearly, the theorem prover did

The third observation concerns the time taken to carry out the proof. Once a vague outline of a proof ha

The fourth observation is an incidental one that the entire formalization and proof was carried out in a s

The fifth observation deals with the use of the de Bruijn notation. The reader might suspect that this was

The second approach is to define the operation of substitution so that bound variables are automatically

Finally, the de Bruijn notation does make the proof a lot easier mainly because it is a better notation for

Some readers might find it disturbing that the representation for λ-terms should have made such a big di

6.3 Acknowledgements

The work reported in this paper was supervised by Bob Boyer and J Moore. They suggested the proble

Appendix A

Selections from Proofs generated by the Boyer-Moore Theorem Prover

A.1 The Proof of SUBST-SUBST

```
(PROVE-LEMMA SUBST-SUBST
             (REWRITE)
             (IMPLIES (AND (NUMBERP M) (LESSP M N))
                      (EQUAL (SUBST (SUBST X (BUMP Z M) (ADD1 N))
                                    (SUBST Y Z N)
                                    (ADD1 M))
                             (SUBST (SUBST X Y (ADD1 M)) Z N)))
             ((INDUCT (SUBST-INDUCT X Y Z M N))))
This conjecture can be simplified, using the abbreviations IMPLIES, NOT, OR,
and AND, to three new formulas:

Case 3. (IMPLIES (AND (LAMBDAP X)
                      (IMPLIES (AND (NUMBERP (ADD1 M))
                                    (LESSP (ADD1 M) (ADD1 N)))
                               (EQUAL (SUBST (SUBST (BODY X)
                                                    (BUMP (BUMP Z 0) (ADD1 M))
                                                    (ADD1 (ADD1 N)))
                                             (SUBST (BUMP Y 0)
                                                    (BUMP Z 0)
                                                    (ADD1 N))
                                             (ADD1 (ADD1 M)))
                                      (SUBST (SUBST (BODY X)
                                                    (BUMP Y 0)
                                                    (ADD1 (ADD1 M)))
                                             (BUMP Z 0)
                                             (ADD1 N))))
                      (NUMBERP M)
                      (LESSP M N))
                 (EQUAL (SUBST (SUBST X (BUMP Z M) (ADD1 N))
                               (SUBST Y Z N)
                               (ADD1 M))
                        (SUBST (SUBST X Y (ADD1 M)) Z N))),

  which simplifies, using linear arithmetic, rewriting with the lemmas
  SUB1-ADD1, BUMP-BUMP, BUMP-SUBST, SUBST-LAMBDA, LAMBDAP-SUBST, and
  BODY-LAMBDA, and opening up the functions LESSP, AND, IMPLIES, and SUBST,
  to:

        T.

Case 2. (IMPLIES
              (AND (NOT (LAMBDAP X))
                   (COMBP X)
                   (IMPLIES (AND (NUMBERP M) (LESSP M N))
                            (EQUAL (SUBST (SUBST (RIGHT X) (BUMP Z M)
                                                 (ADD1 N))
                                          (SUBST Y Z N)
                                          (ADD1 M))
                                   (SUBST (SUBST (RIGHT X) Y (ADD1 M))
                                          Z N)))
                   (IMPLIES (AND (NUMBERP M) (LESSP M N))
                            (EQUAL (SUBST (SUBST (LEFT X) (BUMP Z M)
```

```
                                          (ADD1 N))
                                    (SUBST Y Z N)
                                    (ADD1 M))
                              (SUBST (SUBST (LEFT X) Y (ADD1 M))
                                    Z N)))
          (NUMBERP M)
          (LESSP M N))
     (EQUAL (SUBST (SUBST X (BUMP Z M) (ADD1 N))
                   (SUBST Y Z N)
                   (ADD1 M))
            (SUBST (SUBST X Y (ADD1 M)) Z N))).
```

This simplifies, rewriting with the lemmas RIGHT-COMB and LEFT-COMB, and opening up the definitions of AND, IMPLIES, and SUBST, to:

```
T.
```

```
Case 1. (IMPLIES (AND (NOT (LAMBDAP X))
                      (NOT (COMBP X))
                      (NUMBERP M)
                      (LESSP M N))
                 (EQUAL (SUBST (SUBST X (BUMP Z M) (ADD1 N))
                               (SUBST Y Z N)
                               (ADD1 M))
                        (SUBST (SUBST X Y (ADD1 M)) Z N))).
```

This simplifies, applying SUB1-ADD1, and expanding the definitions of LESSP and SUBST, to the following 17 new conjectures:

[Most of these 17 cases are trivial and are handled by the linear arithmetic package. Only the interesting cases are displayed below.]

```
.
.
.
```

```
Case 1.13.
        (IMPLIES (AND (NOT (LAMBDAP X))
                      (NOT (COMBP X))
                      (NUMBERP M)
                      (LESSP M N)
                      (NOT (EQUAL X 0))
                      (NUMBERP X)
                      (NOT (EQUAL X (ADD1 M)))
                      (LESSP M (SUB1 X))
                      (EQUAL X (ADD1 N)))
                 (EQUAL (SUBST (BUMP Z M)
                               (SUBST Y Z N)
                               (ADD1 M))
                        (SUBST (SUB1 X) Z N))).
```

But this again simplifies, rewriting with the lemmas ADD1-EQUAL, SUB1-TYPE-RESTRICTION, SUB1-ADD1, and SUBST-NOT-FREE-IN, and expanding the functions SUB1, EQUAL, LESSP, and SUBST, to:

```
        (IMPLIES (AND (NUMBERP M)
                      (LESSP M N)
                      (NUMBERP N)
                      (NOT (EQUAL N M))
                      (EQUAL N 0))
                 (EQUAL Z 0)).
```

But this again simplifies, using linear arithmetic, to:

T.

.
.
.

Case 1.10.

```
(IMPLIES (AND (NOT (LAMBDAP X))
              (NOT (COMBP X))
              (NUMBERP M)
              (LESSP M N)
              (NOT (EQUAL X 0))
              (NOT (NUMBERP X)))
         (EQUAL (SUBST X (SUBST Y Z N) (ADD1 M))
                (SUBST X Z N))).
```

But this again simplifies, opening up SUBST, to:

T.

.
.
.

Case 1.3.

```
(IMPLIES (AND (NOT (LAMBDAP X))
              (NOT (COMBP X))
              (NUMBERP M)
              (LESSP M N)
              (NOT (EQUAL X 0))
              (NUMBERP X)
              (EQUAL X (ADD1 M))
              (NOT (EQUAL X (ADD1 N)))
              (NUMBERP N)
              (LEQ (SUB1 X) N))
         (EQUAL (SUBST X (SUBST Y Z N) (ADD1 M))
                (SUBST Y Z N))).
```

This simplifies again, applying ADD1-EQUAL and SUB1-ADD1, and unfolding SUBST, to:

T.

.
.
.

Case 1.1.

```
(IMPLIES (AND (NOT (LAMBDAP X))
              (NOT (COMBP X))
              (NUMBERP M)
              (LESSP M N)
              (EQUAL X 0))
         (EQUAL (SUBST 0 (SUBST Y Z N) (ADD1 M))
                (SUBST 0 Z N))),
```

which we again simplify, unfolding the definitions of LAMBDAP, COMBP, EQUAL, and SUBST, to:

T.

```
Q.E.D.


[ 13.4 3.1 ]

SUBST-SUBST
```

A.2 The Proof of the Substitutivity of Walks

```
(PROVE-LEMMA WALK-SUBST
             (REWRITE)
             (IMPLIES (NOT (ZEROP N))
                      (EQUAL (WALK (SUB-WALK W M U N)
                                   (SUBST M X N))
                             (SUBST (WALK W M) (WALK U X) N)))
             ((INDUCT (WALK-SUBST-IND W M U X N))))
This formula can be simplified, using the abbreviations ZEROP, IMPLIES, NOT,
OR, AND, LEFT-INSTRS, RIGHT-INSTRS, COMMAND, and BUMP-WALK, to four new
conjectures:

Case 4. (IMPLIES
              (AND (LAMBDAP M)
                   (IMPLIES (NOT (ZEROP (ADD1 N)))
                            (EQUAL (WALK (SUB-WALK W (BODY M) U (ADD1 N))
                                         (SUBST (BODY M) (BUMP X 0) (ADD1 N)))
                                   (SUBST (WALK W (BODY M))
                                          (BUMP (WALK U X) 0)
                                          (ADD1 N))))
                   (NOT (EQUAL N 0))
                   (NUMBERP N))
              (EQUAL (WALK (SUB-WALK W M U N)
                           (SUBST M X N))
                     (SUBST (WALK W M) (WALK U X) N))).

  This simplifies, rewriting with the lemma BODY-LAMBDA, and unfolding ZEROP,
  NOT, IMPLIES, SUB-WALK, SUBST, and WALK, to:

        T.

Case 3. (IMPLIES (AND (NOT (LAMBDAP M))
                      (COMBP M)
                      (EQUAL (CADDR W) 'REDUCE)
                      (LAMBDAP (LEFT M))
                      (IMPLIES (NOT (ZEROP N))
                               (EQUAL (WALK (SUB-WALK (CADR W) (RIGHT M) U N)
                                            (SUBST (RIGHT M) X N))
                                      (SUBST (WALK (CADR W) (RIGHT M))
                                             (WALK U X)
                                             N)))
                      (IMPLIES (NOT (ZEROP N))
                               (EQUAL (WALK (SUB-WALK (CAR W) (LEFT M) U N)
                                            (SUBST (LEFT M) X N))
                                      (SUBST (WALK (CAR W) (LEFT M))
                                             (WALK U X)
                                             N)))
                      (NOT (EQUAL N 0))
                      (NUMBERP N))
                 (EQUAL (WALK (SUB-WALK W M U N)
                              (SUBST M X N))
                        (SUBST (WALK W M) (WALK U X) N))),
```

```
  which we simplify, using linear arithmetic, applying the lemmas SUBST-SUBST,
  RIGHT-COMB, SUBST-LAMBDA, LAMBDA-WALK, CAR-CONS, LAMBDAP-SUBST, LEFT-COMB,
  and CDR-CONS, and expanding the definitions of ZEROP, NOT, IMPLIES,
  RIGHT-INSTRS, LEFT-INSTRS, EQUAL, COMMAND, SUB-WALK, SUBST, ADD1, CAR, and
  WALK, to:

        T.

Case 2. (IMPLIES (AND (NOT (LAMBDAP M))
                      (COMBP M)
                      (NOT (AND (EQUAL (CADDR W) 'REDUCE)
                                (LAMBDAP (LEFT M))))
                      (IMPLIES (NOT (ZEROP N))
                               (EQUAL (WALK (SUB-WALK (CADR W) (RIGHT M) U N)
                                            (SUBST (RIGHT M) X N))
                                      (SUBST (WALK (CADR W) (RIGHT M))
                                             (WALK U X)
                                             N)))
                      (IMPLIES (NOT (ZEROP N))
                               (EQUAL (WALK (SUB-WALK (CAR W) (LEFT M) U N)
                                            (SUBST (LEFT M) X N))
                                      (SUBST (WALK (CAR W) (LEFT M))
                                             (WALK U X)
                                             N)))
                      (NOT (EQUAL N 0))
                      (NUMBERP N))
                 (EQUAL (WALK (SUB-WALK W M U N)
                              (SUBST M X N))
                        (SUBST (WALK W M) (WALK U X) N))).

  This simplifies, rewriting with the lemmas RIGHT-COMB, LEFT-COMB, CAR-CONS,
  and CDR-CONS, and unfolding AND, ZEROP, NOT, IMPLIES, RIGHT-INSTRS,
  LEFT-INSTRS, COMMAND, SUB-WALK, SUBST, EQUAL, CAR, and WALK, to:

        T.

Case 1. (IMPLIES (AND (NOT (LAMBDAP M))
                      (NOT (COMBP M))
                      (NOT (EQUAL N 0))
                      (NUMBERP N))
                 (EQUAL (WALK (SUB-WALK W M U N)
                              (SUBST M X N))
                        (SUBST (WALK W M) (WALK U X) N))),

  which simplifies, opening up the functions SUB-WALK, SUBST, and WALK, to
  four new formulas:

  Case 1.4.
         (IMPLIES (AND (NOT (LAMBDAP M))
                       (NOT (COMBP M))
                       (NOT (EQUAL N 0))
                       (NUMBERP N)
                       (NOT (EQUAL M 0))
                       (NUMBERP M)
                       (NOT (EQUAL M N))
                       (LESSP N M))
                  (EQUAL (WALK W (SUB1 M)) (SUB1 M))).

    However this again simplifies, expanding the definition of WALK, to:

         T.
```

```
  Case 1.3.
         (IMPLIES (AND (NOT (LAMBDAP M))
                       (NOT (COMBP M))
                       (NOT (EQUAL N 0))
                       (NUMBERP N)
                       (NOT (EQUAL M 0))
                       (NOT (NUMBERP M)))
                  (EQUAL (WALK W M) M)).

    This again simplifies, opening up WALK, to:

         T.

  Case 1.2.
         (IMPLIES (AND (NOT (LAMBDAP M))
                       (NOT (COMBP M))
                       (NOT (EQUAL N 0))
                       (NUMBERP N)
                       (NOT (EQUAL M 0))
                       (NOT (EQUAL M N))
                       (LEQ M N))
                  (EQUAL (WALK W M) M)).

    This simplifies again, unfolding the function WALK, to:

         T.

  Case 1.1.
         (IMPLIES (AND (NOT (LAMBDAP M))
                       (NOT (COMBP M))
                       (NOT (EQUAL N 0))
                       (NUMBERP N)
                       (EQUAL M 0))
                  (EQUAL (WALK W 0) 0)),

    which we again simplify, opening up LAMBDAP, COMBP, WALK, and EQUAL, to:

         T.

Q.E.D.


[ 14.0 0.9 ]

WALK-SUBST
```

A.3 The Proof of the Diamond Property of Walks

```
(PROVE-LEMMA MAIN
             (REWRITE)
             (EQUAL (WALK (MAKE-WALK M U V) (WALK U M))
                    (WALK (MAKE-WALK M V U) (WALK V M)))
             NIL)

     Name the conjecture *1.


     Perhaps we can prove it by induction.  Four inductions are suggested by
terms in the conjecture.  However, they merge into one likely candidate
induction.  We will induct according to the following scheme:
```

```
(AND (IMPLIES (AND (LAMBDAP M) (P (BODY M) U V))
              (P M U V))
     (IMPLIES (AND (NOT (LAMBDAP M))
                   (COMBP M)
                   (AND (EQUAL (COMMAND U) 'REDUCE)
                        (LAMBDAP (LEFT M)))
                   (P (LEFT M)
                      (LEFT-INSTRS U)
                      (LEFT-INSTRS V))
                   (P (RIGHT M)
                      (RIGHT-INSTRS U)
                      (RIGHT-INSTRS V)))
              (P M U V))
     (IMPLIES (AND (NOT (LAMBDAP M))
                   (COMBP M)
                   (NOT (AND (EQUAL (COMMAND U) 'REDUCE)
                             (LAMBDAP (LEFT M))))
                   (AND (EQUAL (COMMAND V) 'REDUCE)
                        (LAMBDAP (LEFT M)))
                   (P (LEFT M)
                      (LEFT-INSTRS U)
                      (LEFT-INSTRS V))
                   (P (RIGHT M)
                      (RIGHT-INSTRS U)
                      (RIGHT-INSTRS V)))
              (P M U V))
     (IMPLIES (AND (NOT (LAMBDAP M))
                   (COMBP M)
                   (NOT (AND (EQUAL (COMMAND U) 'REDUCE)
                             (LAMBDAP (LEFT M))))
                   (NOT (AND (EQUAL (COMMAND V) 'REDUCE)
                             (LAMBDAP (LEFT M))))
                   (P (LEFT M)
                      (LEFT-INSTRS U)
                      (LEFT-INSTRS V))
                   (P (RIGHT M)
                      (RIGHT-INSTRS U)
                      (RIGHT-INSTRS V)))
              (P M U V))
     (IMPLIES (AND (NOT (LAMBDAP M))
                   (NOT (COMBP M)))
              (P M U V))).
```

Linear arithmetic and the lemmas BODY-LESSP, RIGHT-LESSP, and LEFT-LESSP establish that the measure (COUNT M) decreases according to the well-founded relation LESSP in each induction step of the scheme. Note, however, the inductive instances chosen for V and U. The above induction scheme leads to five new conjectures:

```
Case 5. (IMPLIES (AND (LAMBDAP M)
                      (EQUAL (WALK (MAKE-WALK (BODY M) U V)
                                   (WALK U (BODY M)))
                             (WALK (MAKE-WALK (BODY M) V U)
                                   (WALK V (BODY M)))))
                 (EQUAL (WALK (MAKE-WALK M U V) (WALK U M))
                        (WALK (MAKE-WALK M V U) (WALK V M)))),
```

which we simplify, appealing to the lemma BODY-LAMBDA, and opening up the definitions of MAKE-WALK and WALK, to:

T.

Case 4. (IMPLIES (AND (NOT (LAMBDAP M))

```
                (COMBP M)
                (AND (EQUAL (COMMAND U) 'REDUCE)
                     (LAMBDAP (LEFT M)))
                (EQUAL (WALK (MAKE-WALK (LEFT M)
                                        (LEFT-INSTRS U)
                                        (LEFT-INSTRS V))
                             (WALK (LEFT-INSTRS U) (LEFT M)))
                       (WALK (MAKE-WALK (LEFT M)
                                        (LEFT-INSTRS V)
                                        (LEFT-INSTRS U))
                             (WALK (LEFT-INSTRS V) (LEFT M))))
                (EQUAL (WALK (MAKE-WALK (RIGHT M)
                                        (RIGHT-INSTRS U)
                                        (RIGHT-INSTRS V))
                             (WALK (RIGHT-INSTRS U) (RIGHT M)))
                       (WALK (MAKE-WALK (RIGHT M)
                                        (RIGHT-INSTRS V)
                                        (RIGHT-INSTRS U))
                             (WALK (RIGHT-INSTRS V) (RIGHT M)))))
          (EQUAL (WALK (MAKE-WALK M U V) (WALK U M))
                 (WALK (MAKE-WALK M V U) (WALK V M)))),
```

which we simplify, rewriting with WALK-LAMBDA and WALK-SUBST, and opening up the functions COMMAND, AND, MAKE-WALK, EQUAL, and WALK, to two new formulas:

```
Case 4.2.
      (IMPLIES (AND (COMBP M)
                    (EQUAL (CADDR U) 'REDUCE)
                    (LAMBDAP (LEFT M))
                    (EQUAL (WALK (MAKE-WALK (LEFT M)
                                            (LEFT-INSTRS U)
                                            (LEFT-INSTRS V))
                                 (WALK (LEFT-INSTRS U) (LEFT M)))
                           (WALK (MAKE-WALK (LEFT M)
                                            (LEFT-INSTRS V)
                                            (LEFT-INSTRS U))
                                 (WALK (LEFT-INSTRS V) (LEFT M))))
                    (EQUAL (WALK (MAKE-WALK (RIGHT M)
                                            (RIGHT-INSTRS U)
                                            (RIGHT-INSTRS V))
                                 (WALK (RIGHT-INSTRS U) (RIGHT M)))
                           (WALK (MAKE-WALK (RIGHT M)
                                            (RIGHT-INSTRS V)
                                            (RIGHT-INSTRS U))
                                 (WALK (RIGHT-INSTRS V) (RIGHT M))))
                    (NOT (EQUAL (CADDR V) 'REDUCE)))
               (EQUAL (SUBST (WALK (MAKE-WALK (LEFT M)
                                              (LEFT-INSTRS U)
                                              (LEFT-INSTRS V))
                                   (WALK (LEFT-INSTRS U)
                                         (BODY (LEFT M))))
                             (WALK (MAKE-WALK (RIGHT M)
                                              (RIGHT-INSTRS U)
                                              (RIGHT-INSTRS V))
                                   (WALK (RIGHT-INSTRS U) (RIGHT M)))
                             1)
                      (WALK (CONS (MAKE-WALK (LEFT M)
                                             (LEFT-INSTRS V)
                                             (LEFT-INSTRS U))
                                  (CONS (MAKE-WALK (RIGHT M)
                                                   (RIGHT-INSTRS V)
                                                   (RIGHT-INSTRS U))
```

```
                                   '(REDUCE)))
                            (COMB (WALK (LEFT-INSTRS V) (LEFT M))
                                  (WALK (RIGHT-INSTRS V) (RIGHT M)))))),
```

which again simplifies, applying WALK-LAMBDA, LAMBDA-WALK, RIGHT-COMB, CAR-CONS, LEFT-COMB, and CDR-CONS, and expanding WALK, LEFT-INSTRS, RIGHT-INSTRS, EQUAL, COMMAND, and CAR, to the formula:

```
      (IMPLIES (AND (COMBP M)
                    (EQUAL (CADDR U) 'REDUCE)
                    (LAMBDAP (LEFT M))
                    (EQUAL (LAMBDA (WALK (MAKE-WALK (LEFT M)
                                                    (LEFT-INSTRS U)
                                                    (LEFT-INSTRS V))
                                         (WALK (CAR U) (BODY (LEFT M)))))
                           (WALK (MAKE-WALK (LEFT M)
                                            (LEFT-INSTRS V)
                                            (LEFT-INSTRS U))
                                 (WALK (LEFT-INSTRS V) (LEFT M))))
                    (EQUAL (WALK (MAKE-WALK (RIGHT M)
                                            (RIGHT-INSTRS U)
                                            (RIGHT-INSTRS V))
                                 (WALK (RIGHT-INSTRS U) (RIGHT M)))
                           (WALK (MAKE-WALK (RIGHT M)
                                            (RIGHT-INSTRS V)
                                            (RIGHT-INSTRS U))
                                 (WALK (RIGHT-INSTRS V) (RIGHT M))))
                    (NOT (EQUAL (CADDR V) 'REDUCE)))
               (EQUAL (SUBST (WALK (MAKE-WALK (LEFT M)
                                              (LEFT-INSTRS U)
                                              (LEFT-INSTRS V))
                                   (WALK (CAR U) (BODY (LEFT M))))
                             (WALK (MAKE-WALK (RIGHT M)
                                              (RIGHT-INSTRS U)
                                              (RIGHT-INSTRS V))
                                   (WALK (RIGHT-INSTRS U) (RIGHT M)))
                             1)
                      (SUBST (WALK (MAKE-WALK (LEFT M)
                                              (LEFT-INSTRS V)
                                              (LEFT-INSTRS U))
                                   (WALK (CAR V) (BODY (LEFT M))))
                             (WALK (MAKE-WALK (RIGHT M)
                                              (RIGHT-INSTRS U)
                                              (RIGHT-INSTRS V))
                                   (WALK (RIGHT-INSTRS U) (RIGHT M)))
                             1))),
```

which we further simplify, rewriting with BODY-LAMBDA and LAMBDA-EQUAL, and unfolding the definitions of LEFT-INSTRS, WALK, and RIGHT-INSTRS, to:

```
      T.

Case 4.1.
      (IMPLIES
              (AND (COMBP M)
                   (EQUAL (CADDR U) 'REDUCE)
                   (LAMBDAP (LEFT M))
                   (EQUAL (WALK (MAKE-WALK (LEFT M)
                                           (LEFT-INSTRS U)
                                           (LEFT-INSTRS V))
                                (WALK (LEFT-INSTRS U) (LEFT M)))
                          (WALK (MAKE-WALK (LEFT M)
```

```
                                    (LEFT-INSTRS V)
                                    (LEFT-INSTRS U))
                              (WALK (LEFT-INSTRS V) (LEFT M))))
                  (EQUAL (WALK (MAKE-WALK (RIGHT M)
                                    (RIGHT-INSTRS U)
                                    (RIGHT-INSTRS V))
                              (WALK (RIGHT-INSTRS U) (RIGHT M)))
                         (WALK (MAKE-WALK (RIGHT M)
                                    (RIGHT-INSTRS V)
                                    (RIGHT-INSTRS U))
                              (WALK (RIGHT-INSTRS V) (RIGHT M))))
                  (EQUAL (CADDR V) 'REDUCE))
            (EQUAL (SUBST (WALK (MAKE-WALK (LEFT M)
                                       (LEFT-INSTRS U)
                                       (LEFT-INSTRS V))
                                 (WALK (LEFT-INSTRS U)
                                       (BODY (LEFT M))))
                          (WALK (MAKE-WALK (RIGHT M)
                                       (RIGHT-INSTRS U)
                                       (RIGHT-INSTRS V))
                                 (WALK (RIGHT-INSTRS U) (RIGHT M)))
                          1)
                   (WALK (SUB-WALK (MAKE-WALK (LEFT M)
                                              (LEFT-INSTRS V)
                                              (LEFT-INSTRS U))
                                   (WALK (LEFT-INSTRS V)
                                         (BODY (LEFT M)))
                                   (MAKE-WALK (RIGHT M)
                                              (RIGHT-INSTRS V)
                                              (RIGHT-INSTRS U))
                                   1)
                         (SUBST (WALK (LEFT-INSTRS V) (BODY (LEFT M)))
                                (WALK (RIGHT-INSTRS V) (RIGHT M))
                                1))))).
```

This simplifies again, applying WALK-LAMBDA, LAMBDA-WALK, and WALK-SUBST, and expanding WALK, LEFT-INSTRS, and EQUAL, to:

```
      (IMPLIES (AND (COMBP M)
                    (EQUAL (CADDR U) 'REDUCE)
                    (LAMBDAP (LEFT M))
                    (EQUAL (LAMBDA (WALK (MAKE-WALK (LEFT M)
                                                    (LEFT-INSTRS U)
                                                    (LEFT-INSTRS V))
                                         (WALK (CAR U) (BODY (LEFT M)))))
                           (WALK (MAKE-WALK (LEFT M)
                                            (LEFT-INSTRS V)
                                            (LEFT-INSTRS U))
                                 (WALK (LEFT-INSTRS V) (LEFT M))))
                    (EQUAL (WALK (MAKE-WALK (RIGHT M)
                                            (RIGHT-INSTRS U)
                                            (RIGHT-INSTRS V))
                                 (WALK (RIGHT-INSTRS U) (RIGHT M)))
                           (WALK (MAKE-WALK (RIGHT M)
                                            (RIGHT-INSTRS V)
                                            (RIGHT-INSTRS U))
                                 (WALK (RIGHT-INSTRS V) (RIGHT M))))
                    (EQUAL (CADDR V) 'REDUCE))
             (EQUAL (SUBST (WALK (MAKE-WALK (LEFT M)
                                            (LEFT-INSTRS U)
                                            (LEFT-INSTRS V))
                                 (WALK (CAR U) (BODY (LEFT M))))
```

```
                              (WALK (MAKE-WALK (RIGHT M)
                                               (RIGHT-INSTRS U)
                                               (RIGHT-INSTRS V))
                                    (WALK (RIGHT-INSTRS U) (RIGHT M)))
                              1)
                        (SUBST (WALK (MAKE-WALK (LEFT M)
                                                (LEFT-INSTRS V)
                                                (LEFT-INSTRS U))
                                     (WALK (CAR V) (BODY (LEFT M))))
                               (WALK (MAKE-WALK (RIGHT M)
                                                (RIGHT-INSTRS U)
                                                (RIGHT-INSTRS V))
                                     (WALK (RIGHT-INSTRS U) (RIGHT M)))
                               1))),
```

which further simplifies, applying BODY-LAMBDA and LAMBDA-EQUAL, and unfolding the definitions of LEFT-INSTRS, WALK, and RIGHT-INSTRS, to:

```
      T.
```

```
Case 3. (IMPLIES (AND (NOT (LAMBDAP M))
                      (COMBP M)
                      (NOT (AND (EQUAL (COMMAND U) 'REDUCE)
                                (LAMBDAP (LEFT M))))
                      (AND (EQUAL (COMMAND V) 'REDUCE)
                           (LAMBDAP (LEFT M)))
                      (EQUAL (WALK (MAKE-WALK (LEFT M)
                                              (LEFT-INSTRS U)
                                              (LEFT-INSTRS V))
                                   (WALK (LEFT-INSTRS U) (LEFT M)))
                             (WALK (MAKE-WALK (LEFT M)
                                              (LEFT-INSTRS V)
                                              (LEFT-INSTRS U))
                                   (WALK (LEFT-INSTRS V) (LEFT M))))
                      (EQUAL (WALK (MAKE-WALK (RIGHT M)
                                              (RIGHT-INSTRS U)
                                              (RIGHT-INSTRS V))
                                   (WALK (RIGHT-INSTRS U) (RIGHT M)))
                             (WALK (MAKE-WALK (RIGHT M)
                                              (RIGHT-INSTRS V)
                                              (RIGHT-INSTRS U))
                                   (WALK (RIGHT-INSTRS V) (RIGHT M)))))
                (EQUAL (WALK (MAKE-WALK M U V) (WALK U M))
                       (WALK (MAKE-WALK M V U) (WALK V M)))).
```

This simplifies, applying RIGHT-COMB, WALK-LAMBDA, CAR-CONS, LAMBDA-WALK, LEFT-COMB, CDR-CONS, and WALK-SUBST, and expanding COMMAND, AND, MAKE-WALK, EQUAL, WALK, RIGHT-INSTRS, LEFT-INSTRS, and CAR, to:

```
      (IMPLIES (AND (COMBP M)
                    (NOT (EQUAL (CADDR U) 'REDUCE))
                    (EQUAL (CADDR V) 'REDUCE)
                    (LAMBDAP (LEFT M))
                    (EQUAL (WALK (MAKE-WALK (LEFT M)
                                            (LEFT-INSTRS U)
                                            (LEFT-INSTRS V))
                                 (WALK (LEFT-INSTRS U) (LEFT M)))
                           (WALK (MAKE-WALK (LEFT M)
                                            (LEFT-INSTRS V)
                                            (LEFT-INSTRS U))
                                 (WALK (LEFT-INSTRS V) (LEFT M))))
                    (EQUAL (WALK (MAKE-WALK (RIGHT M)
```

```
                                        (RIGHT-INSTRS U)
                                        (RIGHT-INSTRS V))
                                   (WALK (RIGHT-INSTRS U) (RIGHT M)))
                              (WALK (MAKE-WALK (RIGHT M)
                                        (RIGHT-INSTRS V)
                                        (RIGHT-INSTRS U))
                                   (WALK (RIGHT-INSTRS V) (RIGHT M)))))
          (EQUAL (SUBST (WALK (MAKE-WALK (LEFT M)
                                         (LEFT-INSTRS U)
                                         (LEFT-INSTRS V))
                              (WALK (CAR U) (BODY (LEFT M))))
                        (WALK (MAKE-WALK (RIGHT M)
                                         (RIGHT-INSTRS U)
                                         (RIGHT-INSTRS V))
                              (WALK (RIGHT-INSTRS U) (RIGHT M)))
                        1)
                 (SUBST (WALK (MAKE-WALK (LEFT M)
                                         (LEFT-INSTRS V)
                                         (LEFT-INSTRS U))
                              (WALK (LEFT-INSTRS V)
                                    (BODY (LEFT M))))
                        (WALK (MAKE-WALK (RIGHT M)
                                         (RIGHT-INSTRS U)
                                         (RIGHT-INSTRS V))
                              (WALK (RIGHT-INSTRS U) (RIGHT M)))
                        1))),
```

which again simplifies, appealing to the lemmas WALK-LAMBDA and LAMBDA-WALK, and unfolding WALK and LEFT-INSTRS, to:

```
     (IMPLIES (AND (COMBP M)
                   (NOT (EQUAL (CADDR U) 'REDUCE))
                   (EQUAL (CADDR V) 'REDUCE)
                   (LAMBDAP (LEFT M))
                   (EQUAL (WALK (MAKE-WALK (LEFT M)
                                           (LEFT-INSTRS U)
                                           (LEFT-INSTRS V))
                                (WALK (LEFT-INSTRS U) (LEFT M)))
                          (LAMBDA (WALK (MAKE-WALK (LEFT M)
                                                   (LEFT-INSTRS V)
                                                   (LEFT-INSTRS U))
                                        (WALK (CAR V) (BODY (LEFT M))))))
                   (EQUAL (WALK (MAKE-WALK (RIGHT M)
                                           (RIGHT-INSTRS U)
                                           (RIGHT-INSTRS V))
                                (WALK (RIGHT-INSTRS U) (RIGHT M)))
                          (WALK (MAKE-WALK (RIGHT M)
                                           (RIGHT-INSTRS V)
                                           (RIGHT-INSTRS U))
                                (WALK (RIGHT-INSTRS V) (RIGHT M)))))
          (EQUAL (SUBST (WALK (MAKE-WALK (LEFT M)
                                         (LEFT-INSTRS U)
                                         (LEFT-INSTRS V))
                              (WALK (CAR U) (BODY (LEFT M))))
                        (WALK (MAKE-WALK (RIGHT M)
                                         (RIGHT-INSTRS U)
                                         (RIGHT-INSTRS V))
                              (WALK (RIGHT-INSTRS U) (RIGHT M)))
                        1)
                 (SUBST (WALK (MAKE-WALK (LEFT M)
                                         (LEFT-INSTRS V)
                                         (LEFT-INSTRS U))
```

```
                                    (WALK (CAR V) (BODY (LEFT M))))
                               (WALK (MAKE-WALK (RIGHT M)
                                                (RIGHT-INSTRS U)
                                                (RIGHT-INSTRS V))
                                    (WALK (RIGHT-INSTRS U) (RIGHT M)))
                               1))).
```

However this further simplifies, rewriting with BODY-LAMBDA and LAMBDA-EQUAL, and unfolding the definitions of LEFT-INSTRS, WALK, and RIGHT-INSTRS, to:

T.

```
Case 2. (IMPLIES (AND (NOT (LAMBDAP M))
                      (COMBP M)
                      (NOT (AND (EQUAL (COMMAND U) 'REDUCE)
                                (LAMBDAP (LEFT M))))
                      (NOT (AND (EQUAL (COMMAND V) 'REDUCE)
                                (LAMBDAP (LEFT M))))
                      (EQUAL (WALK (MAKE-WALK (LEFT M)
                                              (LEFT-INSTRS U)
                                              (LEFT-INSTRS V))
                                   (WALK (LEFT-INSTRS U) (LEFT M)))
                             (WALK (MAKE-WALK (LEFT M)
                                              (LEFT-INSTRS V)
                                              (LEFT-INSTRS U))
                                   (WALK (LEFT-INSTRS V) (LEFT M))))
                      (EQUAL (WALK (MAKE-WALK (RIGHT M)
                                              (RIGHT-INSTRS U)
                                              (RIGHT-INSTRS V))
                                   (WALK (RIGHT-INSTRS U) (RIGHT M)))
                             (WALK (MAKE-WALK (RIGHT M)
                                              (RIGHT-INSTRS V)
                                              (RIGHT-INSTRS U))
                                   (WALK (RIGHT-INSTRS V) (RIGHT M)))))
                (EQUAL (WALK (MAKE-WALK M U V) (WALK U M))
                       (WALK (MAKE-WALK M V U) (WALK V M)))).
```

This simplifies, applying RIGHT-COMB, LEFT-COMB, CAR-CONS, and CDR-CONS, and opening up COMMAND, AND, MAKE-WALK, WALK, RIGHT-INSTRS, LEFT-INSTRS, EQUAL, and CAR, to:

T.

```
Case 1. (IMPLIES (AND (NOT (LAMBDAP M))
                      (NOT (COMBP M)))
                (EQUAL (WALK (MAKE-WALK M U V) (WALK U M))
                       (WALK (MAKE-WALK M V U) (WALK V M)))),
```

which simplifies, unfolding the functions MAKE-WALK and WALK, to:

T.

That finishes the proof of *1. Q.E.D.

[28.0 3.0]

MAIN

Appendix B

The List of Definitions and Lemmas

```
1.   (BOOT-STRAP)

2.   Shell Definition.
     Add the shell NLAMBDA of two arguments with
     recognizer NLAMBDAP,
     accessors NBIND and NBODY,
     type restrictions (ONE-OF NUMBERP) and (NONE-OF),
     and default values ZERO and ZERO.

3.   Shell Definition.
     Add the shell NCOMB of two arguments with
     recognizer NCOMBP,
     accessors NLEFT and NRIGHT,
     type restrictions (NONE-OF) and (NONE-OF),
     and default values ZERO and ZERO.

4.   Definition.
     (NSUBST X Y N)
        =
     (IF (NLAMBDAP X)
         (IF (EQUAL (NBIND X) N)
             X
             (NLAMBDA (NBIND X)
                      (NSUBST (NBODY X) Y N)))
         (IF (NCOMBP X)
             (NCOMB (NSUBST (NLEFT X) Y N)
                    (NSUBST (NRIGHT X) Y N))
             (IF (NUMBERP X)
                 (IF (EQUAL X N) Y X)
                 X)))

5.   Definition.
     (NOT-FREE-IN X Y)
        =
     (IF (NLAMBDAP Y)
         (IF (EQUAL X (NBIND Y))
             T
             (NOT-FREE-IN X (NBODY Y)))
         (IF (NCOMBP Y)
             (AND (NOT-FREE-IN X (NLEFT Y))
                  (NOT-FREE-IN X (NRIGHT Y)))
             (NOT (EQUAL X Y))))

6.   Definition.
     (FREE-FOR X Y)
        =
     (IF (NLAMBDAP X)
         (AND (NOT-FREE-IN (NBIND X) Y)
              (FREE-FOR (NBODY X) Y))
         (IF (NCOMBP X)
             (AND (FREE-FOR (NLEFT X) Y)
                  (FREE-FOR (NRIGHT X) Y))
             T))
```

```
7.   Definition.
     (INDEX N LIST)
        =
     (IF (LISTP LIST)
         (IF (EQUAL (CAR LIST) N)
             1
             (ADD1 (INDEX N (CDR LIST))))
         (ADD1 N))

8.   Definition.
     (ALPHA-EQUAL A B X Y)
        =
     (IF (AND (NLAMBDAP A) (NLAMBDAP B))
         (ALPHA-EQUAL (NBODY A)
                      (NBODY B)
                      (CONS (NBIND A) X)
                      (CONS (NBIND B) Y))
         (IF (AND (NCOMBP A) (NCOMBP B))
             (AND (ALPHA-EQUAL (NLEFT A) (NLEFT B) X Y)
                  (ALPHA-EQUAL (NRIGHT A)
                               (NRIGHT B)
                               X Y))
             (IF (AND (NUMBERP A) (NUMBERP B))
                 (EQUAL (INDEX A X) (INDEX B Y))
                 (EQUAL A B)))))

9.   Definition.
     (NTERMP X)
        =
     (IF (NLAMBDAP X)
         (NTERMP (NBODY X))
         (IF (NCOMBP X)
             (AND (NTERMP (NLEFT X))
                  (NTERMP (NRIGHT X)))
             (OR (NUMBERP X) (LITATOM X))))

10.  Definition.
     (NBETA-STEP A B)
        =
     (IF (EQUAL A B)
         T
         (IF (NLAMBDAP A)
             (AND (NLAMBDAP B)
                  (EQUAL (NBIND A) (NBIND B))
                  (NBETA-STEP (NBODY A) (NBODY B)))
             (IF (NCOMBP A)
                 (OR (AND (NLAMBDAP (NLEFT A))
                          (FREE-FOR (NBODY (NLEFT A))
                                    (NRIGHT A))
                          (EQUAL B
                                 (NSUBST (NBODY (NLEFT A))
                                         (NRIGHT A)
                                         (NBIND (NLEFT A)))))
                     (AND (NCOMBP B)
                          (NBETA-STEP (NLEFT A) (NLEFT B))
                          (NBETA-STEP (NRIGHT A) (NRIGHT B))))
                 F)))

11.  Definition.
     (NSTEP A B)
        =
     (OR (ALPHA-EQUAL A B NIL NIL)
         (NBETA-STEP A B))
```

```
12.  Definition.
     (NREDUCTION A B LIST)
        =
     (IF (LISTP LIST)
         (AND (NSTEP (CAR LIST) B)
              (NREDUCTION A (CAR LIST) (CDR LIST)))
         (NSTEP A B))

13.  Shell Definition.
     Add the shell LAMBDA of one argument with
     recognizer LAMBDAP,
     accessor BODY,
     type restriction (NONE-OF),
     and default value ZERO.

14.  Shell Definition.
     Add the shell COMB of two arguments with
     recognizer COMBP,
     accessors LEFT and RIGHT,
     type restrictions (NONE-OF) and (NONE-OF),
     and default values ZERO and ZERO.

15.  Definition.
     (BUMP X N)
        =
     (IF (LAMBDAP X)
         (LAMBDA (BUMP (BODY X) (ADD1 N)))
         (IF (COMBP X)
             (COMB (BUMP (LEFT X) N)
                   (BUMP (RIGHT X) N))
             (IF (LESSP N X) (ADD1 X) X)))

16.  Definition.
     (SUBST X Y N)
        =
     (IF (LAMBDAP X)
         (LAMBDA (SUBST (BODY X) (BUMP Y 0) (ADD1 N)))
         (IF (COMBP X)
             (COMB (SUBST (LEFT X) Y N)
                   (SUBST (RIGHT X) Y N))
             (IF (NOT (ZEROP X))
                 (IF (EQUAL X N)
                     Y
                     (IF (LESSP N X) (SUB1 X) X))
                 X)))

17.  Theorem.  BUMP-BUMP (rewrite):
     (IMPLIES (LEQ N M)
              (EQUAL (BUMP (BUMP Y N) (ADD1 M))
                     (BUMP (BUMP Y M) N)))

18.  Definition.
     (BUMP-SUBST-INDUCT X Y N M)
        =
     (IF (LAMBDAP X)
         (BUMP-SUBST-INDUCT (BODY X)
                            (BUMP Y 0)
                            (ADD1 N)
                            (ADD1 M))
         (IF (COMBP X)
             (AND (BUMP-SUBST-INDUCT (LEFT X) Y N M)
                  (BUMP-SUBST-INDUCT (RIGHT X) Y N M))
             T))
```

```
19.  Theorem.  BUMP-SUBST (rewrite):
     (IMPLIES (LESSP M N)
              (EQUAL (BUMP (SUBST X Y N) M)
                     (SUBST (BUMP X M)
                            (BUMP Y M)
                            (ADD1 N))))
     Hint:  Induct as for (BUMP-SUBST-INDUCT X Y N M).

20.  Definition.
     (SUBST-INDUCT X Y Z M N)
        =
     (IF (LAMBDAP X)
         (SUBST-INDUCT (BODY X)
                       (BUMP Y 0)
                       (BUMP Z 0)
                       (ADD1 M)
                       (ADD1 N))
         (IF (COMBP X)
             (AND (SUBST-INDUCT (LEFT X) Y Z M N)
                  (SUBST-INDUCT (RIGHT X) Y Z M N))
             T))

21.  Theorem.  LAMBDAP-BUMP (rewrite):
     (IMPLIES (LAMBDAP X)
              (LAMBDAP (BUMP X N)))

22.  Theorem.  LAMBDAP-SUBST (rewrite):
     (IMPLIES (LAMBDAP X)
              (LAMBDAP (SUBST X Y N)))

23.  Theorem.  ANOTHER-BUMP-SUBST (rewrite):
     (IMPLIES (LEQ N (ADD1 M))
              (EQUAL (SUBST (BUMP X (ADD1 M)) (BUMP Y M) N)
                     (BUMP (SUBST X Y N) M)))
     Hint:  Induct as for (BUMP-SUBST-INDUCT X Y N M).

24.  Theorem.  BUMP-LAMBDAP (rewrite):
     (IMPLIES (LAMBDAP X)
              (EQUAL (BODY (BUMP X N))
                     (BUMP (BODY X) (ADD1 N))))

25.  Theorem.  SUBST-LAMBDA (rewrite):
     (IMPLIES (LAMBDAP X)
              (EQUAL (BODY (SUBST X Y N))
                     (SUBST (BODY X) (BUMP Y 0) (ADD1 N))))

26.  Definition.
     (LEFT-INSTRS X)
        =
     (CAR X)

27.  Definition.
     (RIGHT-INSTRS X)
        =
     (CADR X)

28.  Definition.
     (COMMAND X)
        =
     (CADDR X)
```

```
29.  Definition.
     (WALK W M)
        =
     (IF (LAMBDAP M)
         (LAMBDA (WALK W (BODY M)))
         (IF (COMBP M)
             (IF (AND (EQUAL (COMMAND W) 'REDUCE)
                      (LAMBDAP (LEFT M)))
                 (SUBST (BODY (WALK (LEFT-INSTRS W) (LEFT M)))
                        (WALK (RIGHT-INSTRS W) (RIGHT M))
                        1)
                 (COMB (WALK (LEFT-INSTRS W) (LEFT M))
                       (WALK (RIGHT-INSTRS W) (RIGHT M))))
             M))

30.  Definition.
     (BUMP-WALK-IND U M N)
        =
     (IF (LAMBDAP M)
         (BUMP-WALK-IND U (BODY M) (ADD1 N))
         (IF (COMBP M)
             (IF (AND (EQUAL (COMMAND U) 'REDUCE)
                      (LAMBDAP (LEFT M)))
                 (AND (BUMP-WALK-IND (LEFT-INSTRS U)
                                     (LEFT M)
                                     N)
                      (BUMP-WALK-IND (RIGHT-INSTRS U)
                                     (RIGHT M)
                                     N))
                 (AND (BUMP-WALK-IND (LEFT-INSTRS U)
                                     (LEFT M)
                                     N)
                      (BUMP-WALK-IND (RIGHT-INSTRS U)
                                     (RIGHT M)
                                     N)))
             T))

31.  Theorem.  LAMBDA-WALK (rewrite):
     (IMPLIES (LAMBDAP M)
              (LAMBDAP (WALK W M)))

32.  Theorem.  BUMP-WALK (rewrite):
     (EQUAL (WALK U (BUMP M N))
            (BUMP (WALK U M) N))
     Hint:  Induct as for (BUMP-WALK-IND U M N).

33.  Theorem.  NLISTP-WALK (rewrite):
     (IMPLIES (NLISTP W)
              (EQUAL (WALK W M) M))
```

```
34.  Definition.
     (SUB-WALK W M U N)
        =
     (IF (LAMBDAP M)
         (SUB-WALK W (BODY M) U (ADD1 N))
         (IF (COMBP M)
             (IF (AND (EQUAL (COMMAND W) 'REDUCE)
                      (LAMBDAP (LEFT M)))
                 (LIST (SUB-WALK (LEFT-INSTRS W)
                                 (LEFT M)
                                 U N)
                       (SUB-WALK (RIGHT-INSTRS W)
                                 (RIGHT M)
                                 U N)
                       'REDUCE)
                 (LIST (SUB-WALK (LEFT-INSTRS W)
                                 (LEFT M)
                                 U N)
                       (SUB-WALK (RIGHT-INSTRS W)
                                 (RIGHT M)
                                 U N)))
             (IF (NOT (ZEROP M))
                 (IF (EQUAL M N) U W)
                 W)))

35.  Definition.
     (WALK-SUBST-IND W M U X N)
        =
     (IF (LAMBDAP M)
         (WALK-SUBST-IND W
                         (BODY M)
                         U
                         (BUMP X 0)
                         (ADD1 N))
         (IF (COMBP M)
             (IF (AND (EQUAL (COMMAND W) 'REDUCE)
                      (LAMBDAP (LEFT M)))
                 (AND (WALK-SUBST-IND (LEFT-INSTRS W)
                                      (LEFT M)
                                      U X N)
                      (WALK-SUBST-IND (RIGHT-INSTRS W)
                                      (RIGHT M)
                                      U X N))
                 (AND (WALK-SUBST-IND (LEFT-INSTRS W)
                                      (LEFT M)
                                      U X N)
                      (WALK-SUBST-IND (RIGHT-INSTRS W)
                                      (RIGHT M)
                                      U X N)))
             T))

36.  Theorem.  SUBST-NOT-FREE-IN (rewrite):
     (EQUAL (SUBST (BUMP X N) Y (ADD1 N))
            X)
     Hint:  Induct as for (SUBST X Y N).

37.  Theorem.  SUBST-SUBST (rewrite):
     (IMPLIES (AND (NUMBERP M) (LESSP M N))
              (EQUAL (SUBST (SUBST X (BUMP Z M) (ADD1 N))
                            (SUBST Y Z N)
                            (ADD1 M))
                     (SUBST (SUBST X Y (ADD1 M)) Z N)))
     Hint:  Induct as for (SUBST-INDUCT X Y Z M N).
```

```
38.  Theorem.  WALK-SUBST (rewrite):
     (IMPLIES (NOT (ZEROP N))
              (EQUAL (WALK (SUB-WALK W M U N)
                           (SUBST M X N))
                     (SUBST (WALK W M) (WALK U X) N)))
     Hint:  Induct as for (WALK-SUBST-IND W M U X N).

39.  Theorem.  WALK-LAMBDA (rewrite):
     (IMPLIES (LAMBDAP X)
              (EQUAL (BODY (WALK U X))
                     (WALK U (BODY X))))

40.  Definition.
     (MAKE-WALK M U V)
        =
     (IF (LAMBDAP M)
         (MAKE-WALK (BODY M) U V)
         (IF (COMBP M)
             (IF (AND (EQUAL (COMMAND U) 'REDUCE)
                      (LAMBDAP (LEFT M)))
                 (SUB-WALK (MAKE-WALK (LEFT M)
                                      (LEFT-INSTRS U)
                                      (LEFT-INSTRS V))
                           (BODY (WALK (LEFT-INSTRS U) (LEFT M)))
                           (MAKE-WALK (RIGHT M)
                                      (RIGHT-INSTRS U)
                                      (RIGHT-INSTRS V))
                           1)
                 (IF (AND (EQUAL (COMMAND V) 'REDUCE)
                          (LAMBDAP (LEFT M)))
                     (LIST (MAKE-WALK (LEFT M)
                                      (LEFT-INSTRS U)
                                      (LEFT-INSTRS V))
                           (MAKE-WALK (RIGHT M)
                                      (RIGHT-INSTRS U)
                                      (RIGHT-INSTRS V))
                           'REDUCE)
                     (LIST (MAKE-WALK (LEFT M)
                                      (LEFT-INSTRS U)
                                      (LEFT-INSTRS V))
                           (MAKE-WALK (RIGHT M)
                                      (RIGHT-INSTRS U)
                                      (RIGHT-INSTRS V)))))
             U))

41.  Theorem.  MAIN (rewrite):
     (EQUAL (WALK (MAKE-WALK M U V) (WALK U M))
            (WALK (MAKE-WALK M V U) (WALK V M)))

42.  Definition.
     (REDUCE W M)
        =
     (IF (LISTP W)
         (REDUCE (CDR W) (WALK (CAR W) M))
         M)

43.  Definition.
     (MAKE-WALK-REDUCE M W V)
        =
     (IF (LISTP V)
         (MAKE-WALK-REDUCE (WALK (CAR V) M)
                           (MAKE-WALK M (CAR V) W)
                           (CDR V))
         W)
```

```
44.  Definition.
     (MAKE-REDUCE-WALK M W V)
        =
     (IF (LISTP V)
         (CONS (MAKE-WALK M W (CAR V))
               (MAKE-REDUCE-WALK (WALK (CAR V) M)
                                 (MAKE-WALK M (CAR V) W)
                                 (CDR V)))
         NIL)

45.  Theorem.  WALK-REDUCE (rewrite):
     (EQUAL (REDUCE (MAKE-REDUCE-WALK M W V)
                    (WALK W M))
            (WALK (MAKE-WALK-REDUCE M W V)
                  (REDUCE V M)))

46.  Definition.
     (MAKE-REDUCE M U V)
        =
     (IF (LISTP U)
         (CONS (MAKE-WALK-REDUCE M (CAR U) V)
               (MAKE-REDUCE (WALK (CAR U) M)
                            (CDR U)
                            (MAKE-REDUCE-WALK M (CAR U) V)))
         U)

47.  Theorem.  LIST-MAKE-REDUCE (rewrite):
     (IMPLIES (LISTP V)
              (EQUAL (REDUCE (MAKE-REDUCE M U V)
                             (REDUCE V M))
                     (REDUCE (MAKE-REDUCE (WALK (CAR V) M)
                                          (MAKE-REDUCE-WALK M (CAR V) U)
                                          (CDR V))
                             (REDUCE V M))))

48.  Theorem.  LIST-MAKE-REDUCE1 (rewrite):
     (EQUAL (REDUCE (MAKE-REDUCE M U (CONS X Y))
                    (REDUCE Y (WALK X M)))
            (REDUCE (MAKE-REDUCE (WALK X M)
                                 (MAKE-REDUCE-WALK M X U)
                                 Y)
                    (REDUCE (CONS X Y) M)))
     Hints:  Consider:
              LIST-MAKE-REDUCE with {V<-(CONS X Y)}
             Enable  LIST-MAKE-REDUCE

49.  Theorem.  CHURCH-ROSSER (rewrite):
     (EQUAL (REDUCE (MAKE-REDUCE M U V)
                    (REDUCE V M))
            (REDUCE (MAKE-REDUCE M V U)
                    (REDUCE U M)))

50.  Definition.
     (TRANSLATE X BOUNDS)
        =
     (IF (NLAMBDAP X)
         (LAMBDA (TRANSLATE (NBODY X)
                            (CONS (NBIND X) BOUNDS)))
         (IF (NCOMBP X)
             (COMB (TRANSLATE (NLEFT X) BOUNDS)
                   (TRANSLATE (NRIGHT X) BOUNDS))
             (IF (NUMBERP X) (INDEX X BOUNDS) X)))
```

```
51.  Theorem.  ALPHA-TRANSLATE (rewrite):
     (IMPLIES (ALPHA-EQUAL A B X Y)
              (EQUAL (TRANSLATE A X)
                     (TRANSLATE B Y)))

52.  Definition.
     (INSERT X BOUNDS N)
        =
     (IF (ZEROP N)
         (CONS X BOUNDS)
         (IF (LISTP BOUNDS)
             (CONS (CAR BOUNDS)
                   (INSERT X (CDR BOUNDS) (SUB1 N)))
             (CONS X NIL)))

53.  Definition.
     (NSUBST-IND X Y Z BOUNDS N)
        =
     (IF (NLAMBDAP X)
         (IF (EQUAL (NBIND X) Z)
             T
             (NSUBST-IND (NBODY X)
                         Y Z
                         (CONS (NBIND X) BOUNDS)
                         (ADD1 N)))
         (IF (NCOMBP X)
             (AND (NSUBST-IND (NLEFT X) Y Z BOUNDS N)
                  (NSUBST-IND (NRIGHT X) Y Z BOUNDS N))
             T))

54.  Definition.
     (TRANS-INSERT-IND X Z BOUNDS N)
        =
     (IF (NLAMBDAP X)
         (TRANS-INSERT-IND (NBODY X)
                           Z
                           (CONS (NBIND X) BOUNDS)
                           (ADD1 N))
         (IF (NCOMBP X)
             (AND (TRANS-INSERT-IND (NLEFT X)
                                    Z BOUNDS N)
                  (TRANS-INSERT-IND (NRIGHT X)
                                    Z BOUNDS N))
             T))

55.  Definition.
     (PRECEDE X BOUNDS N)
        =
     (IF (ZEROP N)
         F
         (IF (LISTP BOUNDS)
             (OR (EQUAL (CAR BOUNDS) X)
                 (PRECEDE X (CDR BOUNDS) (SUB1 N)))
             F))

56.  Theorem.  NPRECEDE-INDEX (rewrite):
     (IMPLIES (AND (NOT (PRECEDE Z BOUNDS N))
                   (LEQ N (LENGTH BOUNDS)))
              (EQUAL (INDEX Z (INSERT Z BOUNDS N))
                     (ADD1 N)))
```

```
57.  Theorem.  PRECEDE-INDEX1 (rewrite):
     (IMPLIES (AND (LESSP N (INDEX X BOUNDS))
                   (LEQ N (LENGTH BOUNDS)))
              (EQUAL (INDEX X (INSERT Z BOUNDS N))
                     (IF (EQUAL X Z)
                         (ADD1 N)
                         (ADD1 (INDEX X BOUNDS)))))

58.  Theorem.  PRECEDE-INDEX (rewrite):
     (IMPLIES (PRECEDE Z BOUNDS N)
              (LESSP (INDEX Z (INSERT Z BOUNDS N))
                     (ADD1 N)))

59.  Theorem.  PRECEDE-INDEX2 (rewrite):
     (IMPLIES (AND (LEQ (INDEX X BOUNDS) N)
                   (LEQ N (LENGTH BOUNDS)))
              (EQUAL (INDEX X (INSERT Z BOUNDS N))
                     (INDEX X BOUNDS)))

60.  Theorem.  TRANSLATE-INSERT (rewrite):
     (IMPLIES (AND (PRECEDE Z BOUNDS N)
                   (LEQ N (LENGTH BOUNDS))
                   (NTERMP X))
              (EQUAL (TRANSLATE X (INSERT Z BOUNDS N))
                     (BUMP (TRANSLATE X BOUNDS) N)))
     Hint:  Induct as for (TRANS-INSERT-IND X Z BOUNDS N).

61.  Theorem.  SUBST-NSUBST (rewrite):
     (IMPLIES (AND (PRECEDE Z BOUNDS N)
                   (NTERMP X)
                   (LEQ N (LENGTH BOUNDS)))
              (EQUAL (SUBST (TRANSLATE X (INSERT Z BOUNDS N))
                            Y
                            (ADD1 N))
                     (TRANSLATE X BOUNDS)))

62.  Definition.
     (NOT-FREE-IND Y Z BOUNDS N)
        =
     (IF (NLAMBDAP Y)
         (NOT-FREE-IND (NBODY Y)
                       Z
                       (CONS (NBIND Y) BOUNDS)
                       (ADD1 N))
         (IF (NCOMBP Y)
             (AND (NOT-FREE-IND (NLEFT Y) Z BOUNDS N)
                  (NOT-FREE-IND (NRIGHT Y) Z BOUNDS N))
             T))

63.  Theorem.  TRANSLATE-INSERT1 (rewrite):
     (IMPLIES (AND (PRECEDE Z (CONS X BOUNDS) (ADD1 N))
                   (NTERMP Y)
                   (LEQ N (LENGTH BOUNDS)))
              (EQUAL (TRANSLATE Y
                                (CONS X (INSERT Z BOUNDS N)))
                     (BUMP (TRANSLATE Y (CONS X BOUNDS))
                           (ADD1 N))))
     Hints:  Consider:
              TRANSLATE-INSERT with {BOUNDS<-(CONS X BOUNDS), N<-(ADD1 N),
                                     X<-Y}
             Enable TRANSLATE-INSERT
```

```
64.  Theorem.  NOT-FREE-IN-TRANSLATE (rewrite):
     (IMPLIES (AND (NOT-FREE-IN Z Y)
                   (LEQ N (LENGTH BOUNDS))
                   (NTERMP Y))
              (EQUAL (TRANSLATE Y (INSERT Z BOUNDS N))
                     (BUMP (TRANSLATE Y BOUNDS) N)))
     Hint:  Induct as for (NOT-FREE-IND Y Z BOUNDS N).

65.  Theorem.  NOT-FREE-IN-CORR (rewrite):
     (IMPLIES (AND (NTERMP Y) (NOT-FREE-IN Z Y))
              (EQUAL (TRANSLATE Y (CONS Z BOUNDS))
                     (BUMP (TRANSLATE Y BOUNDS) 0)))
     Hints:  Consider:
              NOT-FREE-IN-TRANSLATE with {N<-0}
             Enable NOT-FREE-IN-TRANSLATE

66.  Theorem.  SUBST-NSUBST2 (rewrite):
     (IMPLIES (AND (PRECEDE Z (CONS W BOUNDS) (ADD1 N))
                   (NTERMP X)
                   (LEQ N (LENGTH BOUNDS)))
              (EQUAL (SUBST (TRANSLATE X
                                       (CONS W (INSERT Z BOUNDS N)))
                            Y
                            (ADD1 (ADD1 N)))
                     (TRANSLATE X (CONS W BOUNDS))))
     Hints:  Consider:
              SUBST-NSUBST with {BOUNDS<-(CONS W BOUNDS), N<-(ADD1 N)}
             Enable SUBST-NSUBST

67.  Theorem.  SUBST-NSUBST2-CORR (rewrite):
     (IMPLIES (NTERMP X)
              (EQUAL (SUBST (TRANSLATE X (CONS Z (CONS Z BOUNDS)))
                            Y
                            (ADD1 (ADD1 0)))
                     (TRANSLATE X (CONS Z BOUNDS))))
     Hints:  Consider:
              SUBST-NSUBST2 with {W<-Z, N<-0}
             Enable SUBST-NSUBST2

68.  Theorem.  INDEX-INSERT (rewrite):
     (IMPLIES (AND (NOT (EQUAL X Z))
                   (LEQ N (LENGTH BOUNDS)))
              (NOT (EQUAL (INDEX X (INSERT Z BOUNDS N))
                          (ADD1 N))))

69.  Theorem.  TRANSLATE-PRESERVES-SUBST (rewrite):
     (IMPLIES (AND (FREE-FOR X Y)
                   (NOT (PRECEDE Z BOUNDS N))
                   (LEQ N (LENGTH BOUNDS))
                   (NTERMP X)
                   (NTERMP Y))
              (EQUAL (TRANSLATE (NSUBST X Y Z) BOUNDS)
                     (SUBST (TRANSLATE X (INSERT Z BOUNDS N))
                            (TRANSLATE Y BOUNDS)
                            (ADD1 N))))
     Hint:  Induct as for (NSUBST-IND X Y Z BOUNDS N).
```

```
70.  Theorem.  TRANSLATE-PRESERVES-REDUCTION (rewrite):
     (IMPLIES (AND (NLAMBDAP X)
                   (FREE-FOR (NBODY X) Y)
                   (NTERMP X)
                   (NTERMP Y))
              (EQUAL (TRANSLATE (NSUBST (NBODY X) Y (NBIND X))
                                BOUNDS)
                     (SUBST (BODY (TRANSLATE X BOUNDS))
                            (TRANSLATE Y BOUNDS)
                            1)))
     Hints:  Consider:
              TRANSLATE-PRESERVES-SUBST with {X<-(NBODY X), Z<-(NBIND X),
                                              N<-0}
             Enable TRANSLATE-PRESERVES-SUBST

71.  Definition.
     (BETA-STEP X Y)
        =
     (IF (EQUAL X Y)
         T
         (IF (LAMBDAP X)
             (AND (LAMBDAP Y)
                  (BETA-STEP (BODY X) (BODY Y)))
             (IF (COMBP X)
                 (OR (AND (LAMBDAP (LEFT X))
                          (EQUAL Y
                                 (SUBST (BODY (LEFT X)) (RIGHT X) 1)))
                     (AND (COMBP Y)
                          (BETA-STEP (LEFT X) (LEFT Y))
                          (BETA-STEP (RIGHT X) (RIGHT Y))))
                 F)))

72.  Definition.
     (REDUCTION A B LIST)
        =
     (IF (LISTP LIST)
         (AND (BETA-STEP (CAR LIST) B)
              (REDUCTION A (CAR LIST) (CDR LIST)))
         (BETA-STEP A B))

73.  Definition.
     (MAKE-WALK-STEP X Y)
        =
     (IF (EQUAL X Y)
         NIL
         (IF (LAMBDAP X)
             (MAKE-WALK-STEP (BODY X) (BODY Y))
             (IF (COMBP X)
                 (IF (AND (LAMBDAP (LEFT X))
                          (EQUAL Y
                                 (SUBST (BODY (LEFT X)) (RIGHT X) 1)))
                     '(NIL NIL REDUCE)
                     (LIST (MAKE-WALK-STEP (LEFT X) (LEFT Y))
                           (MAKE-WALK-STEP (RIGHT X) (RIGHT Y))))
                 NIL)))

74.  Theorem.  MAKE-WALK-STEP-WALKS (rewrite):
     (IMPLIES (BETA-STEP A B)
              (EQUAL (WALK (MAKE-WALK-STEP A B) A)
                     B))
```

```
75.  Definition.
     (APPEND X Y)
        =
     (IF (LISTP X)
         (CONS (CAR X) (APPEND (CDR X) Y))
         Y)

76.  Definition.
     (MAKE-REDUCE-REDUCTION A B LIST)
        =
     (IF (LISTP LIST)
         (APPEND (MAKE-REDUCE-REDUCTION A
                                        (CAR LIST)
                                        (CDR LIST))
                 (LIST (MAKE-WALK-STEP (CAR LIST) B)
                       NIL))
         (LIST (MAKE-WALK-STEP A B)))

77.  Theorem.  REDUCE-APPEND (rewrite):
     (EQUAL (REDUCE (APPEND X Y) A)
            (REDUCE Y (REDUCE X A)))

78.  Theorem.  MAKE-REDUCE-REDUCTION-REDUCTION (rewrite):
     (IMPLIES (REDUCTION A B LIST)
              (EQUAL (REDUCE (MAKE-REDUCE-REDUCTION A B LIST)
                             A)
                     B))

79.  Definition.
     (LIST-LAMBDA LIST)
        =
     (IF (LISTP LIST)
         (CONS (LAMBDA (CAR LIST))
               (LIST-LAMBDA (CDR LIST)))
         NIL)

80.  Theorem.  LAMBDA-REDUCTION (rewrite):
     (IMPLIES (REDUCTION A A1 LIST)
              (REDUCTION (LAMBDA A)
                         (LAMBDA A1)
                         (LIST-LAMBDA LIST)))

81.  Definition.
     (MAP-COMB-LEFT A LIST)
        =
     (IF (LISTP LIST)
         (CONS (COMB A (CAR LIST))
               (MAP-COMB-LEFT A (CDR LIST)))
         NIL)

82.  Definition.
     (MAP-COMB-RIGHT A LIST)
        =
     (IF (LISTP LIST)
         (CONS (COMB (CAR LIST) A)
               (MAP-COMB-RIGHT A (CDR LIST)))
         NIL)
```

```
83.  Definition.
     (LIST-COMB A B LIST1 LIST2)
        =
     (IF (LISTP LIST1)
         (IF (LISTP LIST2)
             (CONS (COMB (CAR LIST1) (CAR LIST2))
                   (LIST-COMB A B
                              (CDR LIST1)
                              (CDR LIST2)))
             (MAP-COMB-RIGHT B LIST1))
         (IF (LISTP LIST2)
             (MAP-COMB-LEFT A LIST2)
             NIL))

84.  Theorem.  MAP-COMB-RIGHT-REDUCTION2 (rewrite):
     (IMPLIES (REDUCTION A A1 LIST1)
              (REDUCTION (COMB A B1)
                         (COMB A1 B1)
                         (MAP-COMB-RIGHT B1 LIST1)))

85.  Theorem.  MAP-COMB-RIGHT-REDUCTION (rewrite):
     (IMPLIES (AND (BETA-STEP B B1)
                   (REDUCTION A A1 LIST1))
              (REDUCTION (COMB A B)
                         (COMB A1 B1)
                         (MAP-COMB-RIGHT B LIST1)))

86.  Theorem.  MAP-COMB-LEFT-REDUCTION2 (rewrite):
     (IMPLIES (REDUCTION B B1 LIST2)
              (REDUCTION (COMB A1 B)
                         (COMB A1 B1)
                         (MAP-COMB-LEFT A1 LIST2)))

87.  Theorem.  MAP-COMB-LEFT-REDUCTION (rewrite):
     (IMPLIES (AND (REDUCTION B B1 LIST2)
                   (BETA-STEP A A1))
              (REDUCTION (COMB A B)
                         (COMB A1 B1)
                         (MAP-COMB-LEFT A LIST2)))

88.  Theorem.  LIST-COMB-REDUCTION (rewrite):
     (IMPLIES (AND (REDUCTION A A1 LIST1)
                   (REDUCTION B B1 LIST2))
              (REDUCTION (COMB A B)
                         (COMB A1 B1)
                         (LIST-COMB A B LIST1 LIST2)))

89.  Theorem.  BETA-SUBST (rewrite):
     (BETA-STEP (COMB (LAMBDA A) B)
                (SUBST A B 1))
```

```
90.  Definition.
     (MAKE-REDUCTION-WALK W A)
        =
     (IF (EQUAL (WALK W A) A)
         NIL
         (IF (LAMBDAP A)
             (LIST-LAMBDA (MAKE-REDUCTION-WALK W (BODY A)))
             (IF (COMBP A)
                 (IF (AND (LAMBDAP (LEFT A))
                          (EQUAL (COMMAND W) 'REDUCE))
                     (CONS (SUBST (BODY (WALK (LEFT-INSTRS W)
                                              (LEFT A)))
                                  (WALK (RIGHT-INSTRS W) (RIGHT A))
                                  1)
                           (CONS (COMB (WALK (LEFT-INSTRS W)
                                             (LEFT A))
                                       (WALK (RIGHT-INSTRS W)
                                             (RIGHT A)))
                                 (LIST-COMB (LEFT A)
                                            (RIGHT A)
                                            (MAKE-REDUCTION-WALK
                                              (LEFT-INSTRS W)
                                              (LEFT A))
                                            (MAKE-REDUCTION-WALK
                                              (RIGHT-INSTRS W)
                                              (RIGHT A)))))
                     (LIST-COMB (LEFT A)
                                (RIGHT A)
                                (MAKE-REDUCTION-WALK (LEFT-INSTRS W)
                                                     (LEFT A))
                                (MAKE-REDUCTION-WALK (RIGHT-INSTRS W)
                                                     (RIGHT A))))
                 NIL)))

91.  Theorem.  MAKE-REDUCTION-WALK-STEPS (rewrite):
     (REDUCTION A
                (WALK W A)
                (MAKE-REDUCTION-WALK W A))

92.  Theorem.  REDUCTION-APPEND (rewrite):
     (IMPLIES (AND (REDUCTION A B X)
                   (REDUCTION B C Y))
              (REDUCTION A C (APPEND Y (CONS B X))))

93.  Definition.
     (MAKE-REDUCE-STEPS W A)
        =
     (IF (LISTP W)
         (APPEND (MAKE-REDUCE-STEPS (CDR W)
                                    (WALK (CAR W) A))
                 (CONS (WALK (CAR W) A)
                       (MAKE-REDUCTION-WALK (CAR W) A)))
         NIL)

94.  Theorem.  MAKE-REDUCE-STEPS-STEPS (rewrite):
     (REDUCTION A
                (REDUCE W A)
                (MAKE-REDUCE-STEPS W A))
```

```
95.  Definition.
     (MAKE-W X Y Z LIST1 LIST2)
        =
     (REDUCE (MAKE-REDUCE X
                          (MAKE-REDUCE-REDUCTION X Y LIST1)
                          (MAKE-REDUCE-REDUCTION X Z LIST2))
             (REDUCE (MAKE-REDUCE-REDUCTION X Z LIST2)
                     X))

96.  Definition.
     (MAKE-REDUCTION X Y Z LIST1 LIST2)
        =
     (MAKE-REDUCE-STEPS (MAKE-REDUCE X
                                     (MAKE-REDUCE-REDUCTION X Z LIST2)
                                     (MAKE-REDUCE-REDUCTION X Y LIST1))
                        Y)

97.  Theorem.  THE-REAL-CHURCH-ROSSER (rewrite):
     (IMPLIES (AND (REDUCTION X Y LIST1)
                   (REDUCTION X Z LIST2))
              (AND (REDUCTION Y
                              (MAKE-W X Z Y LIST2 LIST1)
                              (MAKE-REDUCTION X Y Z LIST1 LIST2))
                   (REDUCTION Z
                              (MAKE-W X Y Z LIST1 LIST2)
                              (MAKE-REDUCTION X Z Y LIST2 LIST1))))

98.  Theorem.  BOTH-MAKE-W-ARE-SAME (rewrite):
     (IMPLIES (AND (REDUCTION X Y LIST1)
                   (REDUCTION X Z LIST2))
              (EQUAL (MAKE-W X Y Z LIST1 LIST2)
                     (MAKE-W X Z Y LIST2 LIST1)))
     Hints:  Consider:
              CHURCH-ROSSER with {M<-X, U<-(MAKE-REDUCE-REDUCTION X Y LIST1),
                                  V<-(MAKE-REDUCE-REDUCTION X Z LIST2)}
             Enable CHURCH-ROSSER

99.  Theorem.  NBETA-STEP-TRANSLATES (rewrite):
     (IMPLIES (AND (NTERMP A) (NBETA-STEP A B))
              (BETA-STEP (TRANSLATE A X)
                         (TRANSLATE B X)))

100. Definition.
     (TRANS-LIST X)
        =
     (IF (LISTP X)
         (CONS (TRANSLATE (CAR X) NIL)
               (TRANS-LIST (CDR X)))
         NIL)

101. Theorem.  NTERMP-SUBST (rewrite):
     (IMPLIES (AND (NTERMP X) (NTERMP Y))
              (NTERMP (NSUBST X Y N)))

102. Theorem.  NTERMP-BSTEP (rewrite):
     (IMPLIES (AND (NBETA-STEP A B) (NTERMP A))
              (NTERMP B))
     Hint:  Induct as for (NBETA-STEP A B).

103. Theorem.  NTERMP-ASTEP (rewrite):
     (IMPLIES (AND (ALPHA-EQUAL A B X Y) (NTERMP A))
              (NTERMP B))

104. Theorem.  NTERMP-LIST (rewrite):
     (IMPLIES (AND (NREDUCTION A B LIST) (NTERMP A))
              (NTERMP B))
```

```
105. Theorem.  REDUCTION-TRANSLATES (rewrite):
     (IMPLIES (AND (NTERMP A) (NREDUCTION A B LIST))
              (REDUCTION (TRANSLATE A NIL)
                         (TRANSLATE B NIL)
                         (TRANS-LIST LIST)))

106. Definition.
     (FIND-M X N)
        =
     (IF (LAMBDAP X)
         (FIND-M (BODY X) (ADD1 N))
         (IF (COMBP X)
             (IF (LESSP (FIND-M (LEFT X) N)
                        (FIND-M (RIGHT X) N))
                 (FIND-M (RIGHT X) N)
                 (FIND-M (LEFT X) N))
             (DIFFERENCE X N)))

107. Definition.
     (UNTRANS X M N)
        =
     (IF (LAMBDAP X)
         (NLAMBDA (PLUS M (ADD1 N))
                  (UNTRANS (BODY X) M (ADD1 N)))
         (IF (COMBP X)
             (NCOMB (UNTRANS (LEFT X) M N)
                    (UNTRANS (RIGHT X) M N))
             (IF (LESSP N X)
                 (DIFFERENCE X (ADD1 N))
                 (IF (NOT (ZEROP X))
                     (ADD1 (PLUS M (DIFFERENCE N X)))
                     X)))))

108. Definition.
     (TERMP X)
        =
     (IF (LAMBDAP X)
         (TERMP (BODY X))
         (IF (COMBP X)
             (AND (TERMP (LEFT X))
                  (TERMP (RIGHT X)))
             (OR (NOT (ZEROP X)) (LITATOM X)))))

109. Theorem.  NOT-FREE-IN-UNTRANS (rewrite):
     (IMPLIES (AND (TERMP X)
                   (LEQ (FIND-M X N) M)
                   (LESSP (PLUS M N) Y))
              (NOT-FREE-IN Y (UNTRANS X M N)))

110. Theorem.  FREE-FOR-UNTRANS (rewrite):
     (IMPLIES (AND (TERMP X)
                   (TERMP Y)
                   (LEQ N2 N1)
                   (LEQ (FIND-M Y N2) M))
              (FREE-FOR (UNTRANS X M N1)
                        (UNTRANS Y M N2)))
     Hint:  Induct as for (UNTRANS X M N1).
```

```
111. Definition.
     (TRANS-UNTRANS-IND X BOUNDS M N)
        =
     (IF (NLAMBDAP X)
         (TRANS-UNTRANS-IND (NBODY X)
                            (CONS (NBIND X) BOUNDS)
                            M
                            (ADD1 N))
         (IF (NCOMBP X)
             (AND (TRANS-UNTRANS-IND (NLEFT X)
                                     BOUNDS M N)
                  (TRANS-UNTRANS-IND (NRIGHT X)
                                     BOUNDS M N))
             T))

112. Definition.
     (BINDINGS M N)
        =
     (IF (ZEROP N)
         NIL
         (CONS (PLUS M N)
               (BINDINGS M (SUB1 N))))

113. Theorem.  TRANS-UNTRANS (rewrite):
     (IMPLIES (AND (EQUAL (LENGTH BOUNDS) N)
                   (NTERMP X)
                   (LEQ (FIND-M (TRANSLATE X BOUNDS) N)
                        M))
              (ALPHA-EQUAL X
                           (UNTRANS (TRANSLATE X BOUNDS) M N)
                           BOUNDS
                           (BINDINGS M N)))
     Hints:  Disable ALPHA-TRANSLATE
             Induct as for (TRANS-UNTRANS-IND X BOUNDS M N).

114. Theorem.  TERMP-NTERMP (rewrite):
     (IMPLIES (NTERMP X)
              (TERMP (TRANSLATE X BOUNDS)))

115. Theorem.  NTERMP-TERMP (rewrite):
     (IMPLIES (TERMP X)
              (NTERMP (UNTRANS X M N)))

116. Theorem.  INDEX-FREEVAR (rewrite):
     (IMPLIES (AND (LESSP N X)
                   (LEQ (DIFFERENCE X N) M))
              (EQUAL (INDEX (DIFFERENCE (SUB1 X) N)
                            (BINDINGS M N))
                     X))

117. Theorem.  UNTRANS-TRANS (rewrite):
     (IMPLIES (AND (TERMP X) (LEQ (FIND-M X N) M))
              (EQUAL (TRANSLATE (UNTRANS X M N)
                                (BINDINGS M N))
                     X))
```

```
118. Theorem.  UNTRANS-SUBST (rewrite):
     (IMPLIES (AND (TERMP X)
                   (TERMP Y)
                   (LAMBDAP X)
                   (LEQ (FIND-M Y N) M)
                   (LEQ (FIND-M X N) M))
              (EQUAL (TRANSLATE (NSUBST (NBODY (UNTRANS X M N))
                                        (UNTRANS Y M N)
                                        (NBIND (UNTRANS X M N)))
                                (BINDINGS M N))
                     (SUBST (BODY X) Y 1)))
     Hints:  Consider:
              TRANSLATE-PRESERVES-REDUCTION with
              {X<-(UNTRANS X M N), Y<-(UNTRANS Y M N), BOUNDS<-(BINDINGS M N)}
             Enable TRANSLATE-PRESERVES-REDUCTION

119. Theorem.  LITATOM-TRANSLATE (rewrite):
     (IMPLIES (LITATOM (TRANSLATE A BOUNDS))
              (EQUAL (TRANSLATE A BOUNDS) A))

120. Theorem.  TRANSLATE-ALPHA (rewrite):
     (IMPLIES (AND (NTERMP A)
                   (NTERMP B)
                   (EQUAL (TRANSLATE A BOUNDS)
                          (TRANSLATE B BOUNDS2)))
              (ALPHA-EQUAL A B BOUNDS BOUNDS2))
     Hints:  Induct as for (ALPHA-EQUAL A B BOUNDS BOUNDS2).
             Consider:
              LITATOM-TRANSLATE
              LITATOM-TRANSLATE with {A<-B, BOUNDS<-BOUNDS2}
             Enable LITATOM-TRANSLATE and LITATOM-TRANSLATE

121. Theorem.  TERMP-BUMP (rewrite):
     (IMPLIES (TERMP X) (TERMP (BUMP X N)))

122. Theorem.  TERMP-SUBST (rewrite):
     (IMPLIES (AND (TERMP X)
                   (TERMP Y)
                   (NOT (ZEROP N)))
              (TERMP (SUBST X Y N)))

123. Definition.
```