8/15/2003

AUTOMATION OF EQUATIONAL PROOF VERIFICATION USING HASKELL

by Sam Lai

Under the Supervision of Dr. Hamilton Richards

Department of Computer Sciences
The University of Texas at Austin

Abstract

Mistakes in a proof are strictly intolerable, because proofs are meant to assure the correctness of a theorem. Many proofs are still derived by humans today; consequently, they are error-prone. In order to minimize human errors, many rigorous proof styles have been developed, and equational logic is one of them. However, when writing an equational proof, it is still possible to make errors. Thus, it would be helpful to have software that checks proofs. This paper presents a logical model and a Haskell implementation for checking equational proofs in propositional logic. After reading the paper, the reader should be able to recognize the main obstacles to be overcome in building a proof verification tool in Haskell.

TABLE OF CONTENTS

1. Introduction 1
1.1. Equational Logic 1
1.2. Pitfalls 3
2. Logical Model 4
2.1. Definition of a Correct Proof 5
2.2. Error Detection 5
2.2.1. Detecting Syntactic Errors 5
2.2.2. Detecting Logical Errors 5
3. Concrete Implementation 9
3.1. Client-Server Overview 10
3.2. The Server 11
3.3. The Client 12
3.4. The Verification Implementation 13
4. Conclusion 17
5. Future Work 17
6. References 17

1. Introduction

There are many ways to write a proof, some of them formal and some informal. Informal proofs involve the use of natural languages, such as English, which are well known for their ambiguity. In addition, natural languages do not have rigorous syntax rules; therefore, it is extremely difficult to parse and verify proofs written in natural languages. Formal proofs, in contrast, enforce strict grammar rules, which not only reduce the chance of mistakes, but more importantly, make the proof verification process more feasible.

1.1. Equational Logic

Equational logic, a type of formal logic, should be fairly familiar to most people, because it is very similar to equation manipulation in algebra. The equational logic presented in this paper is the one described by Gries and Schneider [1]. Their equational logic is based on equality and Leibniz's rule (defined next) for substitution of equals for equals. Leibniz's rule states that if expression X equals expression Y, then the result of replacing variable z in an expression E with X would be the same as replacing z with Y.

One part of our equational logic is a set of axioms, which are Boolean expressions that are postulated to be true. Axioms are usually validated by a truth table (see Example 1). The other part of our equational logic consists of three inference rules: Leibniz, Transitivity, and Substitution (presented in Table 1).

Example 1 – Validate Axioms

Axiom, Associativity of ≡: ((p ≡ q) ≡ r) ≡ (p ≡ (q ≡ r))

p	q	r	((p ≡ q) ≡ r)	(p ≡ (q ≡ r))
true	true	true	true	true
true	true	false	false	false
true	false	true	false	false
true	false	false	true	true
false	true	true	false	false
false	true	false	true	true
false	false	true	true	true
false	false	false	false	false

Axiom, Symmetry of ≡: p ≡ q ≡ q ≡ p

p	q	p ≡ q ≡ q
true	true	true
true	false	true
false	true	false
false	false	false

Table 1 – Fundamental Inference Rules

Leibniz: X = Y ⇒ E[z := X] = E[z := Y]
Transitivity: X = Y ∧ Y = Z ⇒ X = Z
Substitution: X ⇒ X[z := Y]

A theorem is either (i) an axiom, (ii) the conclusion of an inference rule whose premises are theorems, or (iii) a Boolean expression that, using the inference rules, is proved equal to an axiom or a previously proved theorem. A Leibniz transformation is a transformation from one expression to another by Leibniz's rule. An equational proof is a sequence of Leibniz transformations, which has the format described in Table 2. Theorems are usually proved to be true by equational proofs (see Example 2).

Table 2 – Equational Proof Format

```
  E0
= <Explanation of why E0 = E1, using Leibniz's rule>
  E1
= <Explanation of why E1 = E2, using Leibniz's rule>
  E2
= <Explanation of why E2 = E3, using Leibniz's rule>
  E3
```

The proof establishes that E0 = E3, by Transitivity and the individual steps E0 = E1, E1 = E2, and E2 = E3.

Example 2 – Validate Theorems

Theorem, `p ≡ p ≡ q ≡ q`

```
  p ≡ p ≡ q ≡ q
= <Symmetry of ≡ - replace p ≡ q ≡ q by p>
  p ≡ p
= <Symmetry of ≡ - replace first p by p ≡ q ≡ q>
  p ≡ q ≡ q ≡ p
```

The final expression is an axiom, namely Symmetry of ≡, so the theorem is proven.

1.2. Pitfalls

There are many kinds of mistakes people can make when writing an equational proof (see Example 3). An error can be categorized into one of the following: syntax errors, logic errors, and assumption errors. A syntax error occurs when there exists some expression that cannot be parsed using the context-free grammar presented in Table 3. A logic error occurs when there is a mistake in Leibniz transformations. Finally, an assumption error occurs when the applied theorem in a Leibniz transformation is not a true expression or not proven yet.

Table 3 – Context-Free Grammar in Extended BNF [3] for Equational Logic

```
<theorem>     ::= <axiom> [<newline> <proof>]

<axiom>       ::= <name> <newline>
                  <expression>
<proof>       ::=    <expression> <newline>
                   = <name> <newline>
                     <expression>
                  {= <name> <newline>
                     <expression>}

<expression> ::= {<expression> (≡|≠)} <imp>

<imp>         ::= <and_or> | <imp> ⇒ <and_or> | <and_or> ⇐ <imp>

<and_or>      ::= {<and_or> (∧|∨)} <not>

<not>         ::= {¬} <token>

<token>       ::= '(' <expression> ')' | <constant> | <variable>

<constant>    ::= true | false

<name>        ::= (<variable>|.|_) {<variable>|.|_}

<variable>    ::= <letter> {(<letter>|<digit>)}

<letter>      ::= a..z | A..Z

<digit>       ::= 0..9
```

Note: White spaces have no significance in this grammar. They are used to help the reader see the visual structure. Also <newline> is a blank line, which is usually represented by ASCII character '\n' in a program.

2. Logical Model

From a logical perspective, a proof verification tool takes a proof as input, and its output indicates whether the given proof is correct. Our logical model specializes the input as an equational proof and the output as an integer, such that if the output is equal to zero, then the proof is correct; otherwise, the output indicates the line number of the error in the proof. Basically, that is how the tool works; now let's discuss how to implement such functionality logically.

2.1. Definition of a Correct Proof

An equational proof is correct if and only if it contains no syntax errors, logic errors, or assumption errors. An equational proof starts with a conjecture. The goal of the proof is to transform the conjecture to a theorem, and then by Transitivity, we can conclude that the conjecture is indeed a theorem. By the same argument, it is also correct to prove backwards, such that the initial expression is a theorem and the final expression is the conjecture. If the conjecture is of the form P ≡ Q, then another alternative is to transform P to Q or Q to P.

2.2. Error Detection

Error detection takes three steps, namely, syntax error detection, logical error detection, and assumption error detection. Parsing the proof to a parse tree can uncover all syntactic errors. Only after the proof is syntactically correct, does it make sense to detect other errors. By recursively verifying each step from the initial expression to the final expression; we can uncover all logical errors. An assumption error can be easily detected, by detecting references to unproven theorems or invalid axioms.

2.2.1.Detecting Syntactic Errors

To detect syntactic errors, we can parse an equational proof to a parse tree. There are a number of ways to parse with context-free grammars in various programming languages, but I will only demonstrate one concrete parsing implementation in Haskell (see Section 3.2). In short, if a proof can be parsed successfully using the grammar in Table 3, then it is syntactically correct; otherwise, there exists some syntax error in the proof.

2.2.2.Detecting Logical Errors

Each transformation in an equational proof is an application of Leibniz's rule, known as a Leibniz transformation. If all Leibniz transformations in the proof are valid and the final expression is a theorem, then the proof is correct. Thus, verifying a Leibniz transformation is the most essential part of the logical error detection.

When verifying a Leibniz transformation, our goal is to identify the place of substitution and the value being substituted for each variable (see Example 3). Let's call the

expression before transformation P, the theorem applied T, and the expression after transformation Q. The verification algorithm is a two-dimensional search. The outer search attempts to find a possible place of substitution z from the permutations of P. The inner search attempts to find a subexpression X of T that can map all its variables to some subexpression of z consistently, where T = X ≡ Y and B is the mapping. Let R be the expression after replacing the subexpression corresponding to z in P with Y. The search stops when the result of replacing variables in R with its bindings from B is consistent to Q or there is no more possible place for substitution.

Example 3 – Leibniz Transformation Verification Steps

Step 1. Identify the Place of Substitution
Step 2. Identify the Value of Substitution

Example:

```
  p ≡ q ≡ r
= <p ≡ q ≡ s>
  s ≡ r

Leibniz's rule: X = Y ⇒ E[z := X] = E[z := Y]
```

Where is the place of substitution?

```
E = z ≡ r
```

What is the value of substitution?

```
X = p ≡ q
Y = s
```

Note: Identifying the place of substitution is the same as identifying z in E.

Identifying the place of substitution may require testing all possible substitutions. However, the range of possibility grows exponentially with the number of equivalence operators, because Symmetry and Associativity axioms for ≡ (shown in Example 1) are

often used within a transformation implicitly to shorten proofs (see Example 4). Thus, the cost of verifying a proof would rise dramatically as the proof becomes more complex.

Example 4 – Proof Comparisons

This example shows two versions of the same proof, one with implicit use of Symmetry and Associativity axioms for ≡ and the other without.

Explicit	Implicit
(q ≡ true) ≡ q ≡ p ≡ p	(q ≡ true) ≡ q ≡ p ≡ p
= <Associativity of ≡>	= <true ≡ q ≡ q>
q ≡ true ≡ q ≡ p ≡ p	p ≡ q ≡ q ≡ p
= <Symmetry of ≡>	
true ≡ q ≡ q ≡ p ≡ p	
= <true ≡ q ≡ q>	
q ≡ q ≡ p ≡ p	
= <Symmetry of ≡>	
q ≡ p ≡ q ≡ p	
= <Symmetry of ≡>	
p ≡ q ≡ q ≡ p	

Clearly, it is necessary to enforce some restrictions in order to reduce the number of possibilities. My solution allows the user to leave some applications of the Symmetry and Associativity axioms implicit, but not all of them. Specifically, when an expression P is transformed to another expression Q using a theorem T, the Symmetry and Associativity axioms can be used implicitly on P but can be used only explicitly on Q (see Example 5).

Example 5 – Proof Comparisons

This example shows two versions of the same proof, one with partially implicit use of Symmetry and Associativity axioms for ≡ and the other with totally implicit use.

Partially Implicit	Totally Implicit
`(q ≡ true) ≡ q ≡ p ≡ p`	`(q ≡ true) ≡ q ≡ p ≡ p`
`= <true ≡ q ≡ q>`	`= <true ≡ q ≡ q>`
`q ≡ q ≡ p ≡ p`	`p ≡ q ≡ q ≡ p`
`= <Symmetry of ≡>`	
`p ≡ q ≡ q ≡ p`	

Let's analyze the first transformation in both versions of the proof. The original expression P in both versions is "`(q ≡ true) ≡ q ≡ p ≡ p`". The applied theorem T in both versions is "`true ≡ q ≡ q`". We want to replace "`(q ≡ true)`" in P with "q" using T, but the parse tree structure of "`(q ≡ true)`" in P does not match the parse tree structure of "`true ≡ q`" in T. Thus, we need to convert the structure of P. Recall that we could apply Symmetry and Associativity axioms implicitly on P in both versions. After doing so, P becomes "`true ≡ q ≡ q ≡ p ≡ p`". The result expression Q for Partially Implicit version is "`q ≡ q ≡ p ≡ p`", but the result expression Q for Totally Implicit version is "`p ≡ q ≡ q ≡ p`". Recall in the Partially Implicit version, we cannot apply any Symmetry and Associativity axioms implicitly on Q, so the extra step in the Partially Implicit version cannot be avoided. Here, we see a clear restriction of the Partially Implicit version.

However, there is an additional problem. Even if we know the place of substitution (the location of ***z*** in ***E*** from Leibniz's rule), it is still unclear which subexpression of P is substituted for which variable in X. Our algorithm tries to generate a sequence of different combinations of X ≡ Y, and for each combination the algorithm attempts to bind the subexpression of P corresponding to ***z*** to variables in X. Once X successfully binds all of its variables, the variables in Y would assume the same bindings as X. If the free variables of Y (variables that are in Y, but not X) are bound consistently with some value to Q after the transformation, then the transformation is correct. The algorithm terminates when there is no more possibility or a valid substitution is found. Example 6 demonstrates the algorithm.

Example 6 – Finding Values of Substitution

```
  a ≡ b ≡ c
= <Symmetry of ≡ (3.2), p ≡ q ≡ q ≡ p>
  d ≡ d ≡ a ≡ c ≡ b
```

Leibniz's rule: $X = Y \Rightarrow E[z := X] = E[z := Y]$

Given that $E = z \equiv b$, where $z = a \equiv c$.
Now, let's break up the theorem applied $p \equiv q \equiv q \equiv p$ into $X \equiv Y$.
Let's consider the case $X = p$ and $Y = q \equiv q \equiv p$.

```
  E[z := X]
= E[z := p]
= p ≡ b
```

Because we want $E = a \equiv b \equiv c$, we have to bind p to $a \equiv c$. q is a free variable, since we only bound p, so we can bind it to anything. In this case, q is bound to d.

```
  E[z := Y]
= E[z := q ≡ q ≡ p]
= q ≡ q ≡ p ≡ b
```

Recall the bindings are p to $a \equiv c$ and q to d.

```
  X = Y
⇒ <Leibniz's rule>
  E[z := X] = E[z := Y]
⇒ <Substitution rule>
  E[z := X][p,q := a ≡ c,d] = E[z := Y][p,q := a ≡ c,d]
= a ≡ b ≡ c = d ≡ d ≡ a ≡ c ≡ b
```

Thus, the transformation is correct.

3. Concrete Implementation

This section presents the architecture of Proof Checker, an implementation of the logical model described. Proof Checker separates implementation from interface [2], so that they

can be implemented in different programming languages and either one can be updated without affecting the other.

Before Proof Checker can verify a proof, it must understand (parse) the proof. Because the Haskell ParseLib library [4] simplifies the process of translating context-free grammars to a set of concrete parsers, I chose to implement the parser in Haskell. Consequently, it was convenient to implement the proof checking logic in Haskell as well.

On the other hand, when building Proof Checker's user interface, the main focus is to increase the user's productivity. Therefore, the interface should be simple to master, without frequent consulting the user manual, so I chose graphical interface over command-prompt interface. Furthermore, I chose to implement the graphical interface in Java Swing, because it is portable to most platforms and easier to be deployed from a web page in the future.

Finally, we need a way to communicate between the implementation and the user interface. I used TCP/IP sockets to bridge them, so they can run on different platforms. The implementation part of Proof Checker takes the role of server and the interface part acts as a client. This design enables upgrades to the server transparent to the user.

3.1. Client-Server Overview

The Proof Checker's user inputs theorems and their proofs to the graphical interface client. After the user establishes a connection with the server, the client sends a service request to the server, which spawns a thread to process the request and sends the result back to the client.

The client program maintains a database of axioms and theorems, which is updated with the results received from the server. Maintaining the database on the client machine instead of the server machine induces heavier network traffic, because all propositions used in a proof has to be transferred along with the proof. Under normal operations, this induced overhead should be relatively small, but if performance ever becomes an issue or offline processing is desired, the server program can be installed on the same machine as the client program.

3.2. The Server

The server, a plain command-line interactive back-end Haskell application, is responsible for accepting and processing the client's proof checking requests. First, the server spawns a thread to listen for client connections. The spawned thread, which is referred to as the **accepting thread** binds the server to a local port, and then waits to accept client connections. When a request arrives, the **accepting thread** buffers the received data until the full content has arrived. When the request is complete, the **accepting thread** spawns another thread to process it. Then, the processing thread passes the buffer, a String list, to a function called **verify**, which performs proof checking on the list. The input list's format is described in Table 4.

Table 4 – Verification Input Format

```
<input_file> ::= {<newline>} <theorem> <newline> [<file>]
```

The function **verify** parses the input String list to a set of axioms and theorems, where each theorem is accompanied with an equational proof. The verification algorithms for axioms and theorems are very different. As mentioned in Section 1.1, to verify an axiom, the program assigns variables in the axiom with all possible value combinations and evaluates the axiom for each combination. If all combinations of variable assignment to the axiom evaluate to true, then the axiom is correct. On the other hand, to verify a theorem, we need to check its equational proof. If the proof is correct, then we must make sure the proof indeed proves the theorem.

In my implementation, Proof Checker generates the parse tree using Haskell's ParseLib library and the unambiguous context-free grammar presented in Table 3. Example 7 shows some examples for translating a context-free grammar to a Parser (prior knowledge of Haskell and the Hutton/Meijer parsing library [4] is assumed).

Example 7 – ParseLib Examples

```
-- <expression> ::= {<expression> (≡|≠)} <imp>
equ_P :: Parser Expression
equ_P = do term <- imp_P
           left_assoc term
           where
           left_assoc x = do string "=="  -- stands for ≡
                             y <- imp_P
                             left_assoc (Exp2 Equ x y)
                          +++
                          do string "/="  -- stands for ≠
                             y <- imp_P
                             left_assoc (Exp2 Not_Equ x y)
                          +++
                             return x

-- <constant> ::= true | false
constant_P :: Parser Expression
constant_P = do string "true"
                return (Const TRUE)
             +++
             do string "false"
                return (Const FALSE)
```

3.3. The Client

The client is a graphical Java Swing application. It communicates with the server to verify axioms and theorems, and it maintains a database based on the results returned by the server. In addition, to be more user friendly, the proofs in the program look identical to the proofs a user would write on a paper. This feature is accomplished by facilitating the input of propositions via special buttons for common equational operators like ≡ that cannot be typed in from a keyboard directly. Note, before a Unicode character like ≡ is sent to the server, the client automatically replaces it with its counterparts described in Table 5. This replacement is necessary, because the server uses ASCII character representations for those operators. Finally, the client automatically arranges axioms and theorems according to the format described in Table 5.

Table 5 – Equational Operator Mappings between Client and Server

Operator	Client	Server
Not	¬	~
And	∧	&&
Or	∨	||
Forward Implication	⇒	=>
Backward Implication	⇐	<=
Equivalent	≡	==
Not Equivalent	≢	/=

A typical proof checking session begins when the user creates or opens a workspace. The workspace stores a set of propositions, each having a unique name in the workspace and containing an option flag indicating whether the proposition should be saved in the database. (Propositions that are not saved in the database are considered lemmas.) After an active workspace is established, the user inputs a set of propositions to be checked by the server. Then the user sends the workspace to the server, and the server would return the result.

The returned result contains a number and a string. If the number is zero, then there is no error, and the string is disregarded. Otherwise, the number indicates the line number of the error occurrence and the string is the error message. The client program only makes changes to the database when there is no error; otherwise, the line number of the error and the error message are reported to the user.

3.4. The Verification Implementation

Recall from Section 2.2 that all logical errors in a proof are uncovered by verifying each Leibniz transformation in an equational proof from the initial expression to the final expression. In my implementation, **chkTransformation** is the function that checks a

Leibniz transformation. To fully explain the function **chkTransformation**, we first introduce functions **mapBinding** and **compatible** first.

Table 6 – Source Code of Function "mapBinding"

```
 1 mapBinding :: Expression -> Expression -> (Bool, [(String, Expression)])
 2 mapBinding form expression
 3   = case (form, expression) of
 4      ((Exp1 Par x),  x')               -> mapBinding x x'
 5      (x, (Exp1 Par x'))                -> mapBinding x x'
 6      ((Exp1 op x), (Exp1 op' x'))
 7        -> if op == op'
 8             then combine (mapBinding x x') (True, [])
 9             else (False, [])
10      ((Exp2 op x y), (Exp2 op' x' y'))
11        -> if op == op'
12             then combine (mapBinding x x') (mapBinding y y')
13             else (False, [])
14      ((Var x), x')                     -> (True, [(x, x')])
15      ((Const x), (Const y))            -> (x==y, [])
16      otherwise                         -> (False, [])
17    where
18    combine::(Bool, [(String, Expression)]) -> (Bool, [(String, Expression)])
19                                            -> (Bool, [(String, Expression)])
20    combine x y = if (fst x) && (fst y)
21                  then case bindings of
22                         Just b  -> (True, b)
23                         Nothing -> (False, [])
24                  else (False, [])
25                  where bindings = merge (snd x) (snd y)
26    merge :: [(String, Expression)] -> [(String, Expression)]
27                                    -> Maybe [(String, Expression)]
28    merge [] y = Just y
29    merge x [] = Just x
30    merge (x:xs) ys = case [p | (p, q) <- ys, (fst x) == p, (snd x) /= q] of
31                        [] -> merge xs (union [x] ys)
32                        otherwise -> Nothing
```

The function **mapBinding** takes two expressions as inputs. The output of the function is a tuple consisting of a Boolean value, indicating whether binding is possible between the two input expressions, and a list of tuples each consisting of a string followed by an expression. The list of tuples represents a set of variable bindings, where the string is a variable name and the expression is the value bound to that variable. Lines 4-5 take away extra parentheses. Lines 6-15 ensure that the structures of the two input expressions are the same; otherwise, they are not bindable (line 16). Finally, the functions **combine** and **merge** are used to integrate individual tuples of binding from line 14 to a single list of such tuples.

Table 7 – Source Code of Function "compatible"

```
 1 compatible :: Expression -> Expression -> [(String, Expression)] -> Bool
 2 compatible form expression bindings
 3   = case (form, expression) of
 4      ((Exp1 Par x), x') -> compatible x x' bindings
 5      (x, (Exp1 Par x')) -> compatible x x' bindings
 6      ((Exp1 op x), (Exp1 op' x')) -> if op == op'
 7                                      then (compatible x x' bindings)
 8                                      else False
 9      ((Exp2 op x y), (Exp2 op' x' y')) -> if op == op'
10                                          then (compatible x x' bindings)
11                                             && (compatible y y' bindings)
12                                          else False
13      ((Var x), x') -> case [q | (p, q) <- bindings, p == x] of
14                         []  -> True
15                         [y] -> y == x'
16      (Const x, Const x') -> x == x'
17      otherwise -> False
```

The function **compatible** takes two expressions and a set of variable bindings as inputs. The output of the function is a Boolean value indicating whether the two expressions are *compatible* with the given bindings. Again, notice in lines 4-5, extra parentheses around an expression are stripped off, so they will not cause any structural differences between the two given expressions. Lines 6-12 ensure the two expressions have the same structure and Lines 13-15 ensure the variable bindings are the same in both expressions; otherwise, they are not compatible (line 17).

Table 8 – Source Code of Function "chkTransformation"

```
 1 chkTransformation :: Expression -> Theorem -> Expression -> Bool
 2 chkTransformation P (Exp2 Equ X Y) Q
 3  = if bindable && (compatible Y Q bindings)
 4      then True
 5      else
 6        if (checkOp P Q)
 7          then let opDegree = case P of
 8                                (Exp1 _ _)   -> 1
 9                                (Exp2 _ _ _) -> 2
10               in
11               or [chkTransformation (child P n) theorem (child Q n)
12                         | n <- [1..opDegree],
13                           n > 0,
14                           and [child P m == child Q m
15                                  | m <- [1..opDegree], m > 0, m /= n]]
16           else False
17        where
18        theorem  = Exp2 Equ X Y
19        mapping  = mapBinding X P
20        bindable = fst mapping
21        bindings = snd mapping
22        checkOp :: Expression -> Expression -> Bool
23        checkOp x y = case (x, y) of
24                        ((Exp1 op _),   (Exp1 op' _))   -> op == op'
25                        ((Exp2 op _ _), (Exp2 op' _ _)) -> op == op'
26                        otherwise -> False
```

The function **chkTransformation** takes an expression P, a theorem T of the form X ≡ Y, and another expression Q as inputs. The output of the function is a Boolean value indicating whether the transformation from P to Q using T is correct. Recall from Section 2.2.2 that a transformation is correct when some sub-expression (***z***) of P can be bound to the variables in X so that P = ***E***[***z***:=X], and Q is *consistent* with ***E***[***z***:=Y], so that (∀v | v∈Y : v∈X⇒v=x ∧ v∉X⇒v=y) where x is the expression bound to v in X and y is some constant expression. This definition of a correct transformation is captured in lines 3-4. The function **chkTransformation** assumes ***z***=P, and it recurses through subexpressions of P until there is a correct transformation or no more sub-expressions (line 11).

4. Conclusion

The goal of this project was to create a model for automating equational proof verification, which could reduce human errors when writing an equational proof. The main obstacle encountered when verifying an equational proof is handling implicit use of Symmetry and Associativity axioms for ≡. The problem is that the number of possible implicit substitutions grows exponentially as the number of ≡ in the expressions of the transformation being verified increases. Thus, it is very inefficient to check for each possible implicit substitution. To solve this problem, I proposed a solution that partially restricts the user from using such implicit substitutions. As a result, verifications of many more proofs are now practical. For a concrete implementation of the logical model presented in this paper, I have written a program called Proof Checker which is capable of verifying equational proofs in propositional logic using only a standard proof method.

5. Future Work

Proof Checker is capable of verifying propositional proofs using only one proof method. There are many other proof methods, such as "Proof by Contradiction" and "Proof by Case Analysis", not yet implemented in Proof Checker, and extending Proof Checker to handle them is my next goal. A much more substantial extension would enable Proof Checker to verify proofs in predicate logic. In addition, my future focuses for the client part are managing theorems using commercial database systems via JDBC and deploying it from a web page.

6. References

[1] D. Gries and F. Schneider, *A Logical Approach to Discrete Math,* New York: Springer-Verlag, 1993.

[2] B. Eckel, *Thinking in Java 2nd Edition*, Prentice Hall PTR, 2000.

[3] R. Sethi, *Programming Languages*, Addison-Wesley Publishing Company, 1996

[4] G. Hutton and E. Meijer, "Monadic Parsing in Haskell (Functional Pearl)", *Journal of Functional Programming*, volume 8, number 4, 1998.