Data Conversion Method between a Natural Number and a Binary Tree for an Inductive Proof and Its Application

Kazuko Takahashi[1], Shizuo Yoshimaru[1]*
and Mizuki Goto[1]

[1] School of Science and Technology, Kwansei Gakuin University
ktaka@kwansei.ac.jp
[2] School of Science and Technology, Kwansei Gakuin University
shizuo.yoshimaru@gmail.com
[3] School of Science and Technology, Kwansei Gakuin University
bub85144@kwansei.ac.jp

Abstract

This paper presents modeling of a binary tree that represents a natural number and gives an inductive proof for its properties using theorem provers. We define a function for converting data from a natural number into a binary tree and give an inductive proof for its well-definedness. We formalize this method, develop a computational model based on it, and apply it to an electronic cash protocol. We also define the payment function on the binary tree and go on to prove the divisibility of electronic cash using the theorem provers Isabelle/HOL and Coq, respectively. Furthermore, we discuss the effectiveness of this method.

1 Introduction

Theorem proving is an important technique to provide a certified system or to ensure bug-free programming. Several theorem provers, which may be called proof assistants, have been developed, including ACL2 [11], PVS [19], Isabelle/HOL [16], Coq [4], Agda [17], and so on. These tools help users to develop formal proofs, either interactively or semi-automatically, typically using induction as the proof strategy. There are numerous applications for such provers. There are relatively few practical applications of these tools in most fields compared to their use in pure mathematics, although there are some outstanding results in microprocessor design [10], C compilers [14], operating system kernels [12], security protocols [1], and secure card systems [3, 13]. To use these provers successfully, it is necessary to construct a suitable model and then select an appropriate proof strategy. In practical applications it may be difficult to form a natural inductive model, and so it may be necessary to build a data structure, which can be difficult for these provers to handle.

We have previously attempted to prove the divisibility of an electronic cash protocol [21]. In this experience, we have encountered difficulties in proving the properties regarding the function that converts a natural number to a binary tree. The binary tree is a complete binary tree, where a Boolean value is attached to each node, and the value of which is defined as a sum of the value of the left and right subtrees. The problem can be reduced to the fact that there exists a function that cannot be defined in a primitive recursive form on an inductively defined data structure. Because a natural number has a linear structure with only one successor, whereas a

[*] Currently, Nochu Information System Co.,LTD.

binary tree has a branching structure with two successors at each branching point, the induction schemes are different, which complicates the proof.

To solve this problem, we introduced a bit sequence as an intermediate data structure, and defined the function from a natural number to a binary tree as a composition of two recursive functions via a bit sequence. Specifically, the first function describes the mapping from a natural number to an intermediate bit sequence and the second function is the mapping from the bit sequence to the binary tree. We formalized this model and associated proof method, and applied it to the divisibility of an electronic cash protocol using Isabelle/HOL. However, this previous work had several drawbacks. First, the model was complicated, because of an intricate labeling for the nodes of a tree; a simpler and more natural model for this specific binary tree has subsequently been identified. Second, the proof was incomplete, as one of the lemmas that relates the bit sequence to the binary tree was left as an axiom. Third, the effectiveness of the method was not discussed; we did not know whether this method could be applied similarly with theorem provers other than Isabelle/HOL, or whether a large number of modifications would be required. In this paper, we provide a simplified model, give complete proofs of the divisibility using the theorem provers Isabelle/HOL and Coq, and discuss the effectiveness of the method.

The rest of this paper is organized as follows. In Section 2, we describe in detail the problems of data conversion and our solution. In Sections 3 and 4, we present a formalization and an inductive proof of the divisibility of an electronic cash protocol. In Section 5, we provide a discussion, and in Section 6, we present our conclusions.

2 Data Conversion Formalization

2.1 Problem

First, we illustrate the problem of converting between a natural number and a binary tree, which are data structures with different induction schemes.

Let NAT and BTREE be a natural number and a binary tree, respectively. NAT is inductively defined with one successor. BTREE is inductively defined in such a form that there are two successors, i.e.,

```
nat_p(0).
nat_p(n) => nat_p(Suc(n)).

tree_p(Tip).
tree_p(lt) & tree_p(rt) => tree_p(Node(_,lt,rt)).
```

where *Suc* denotes the successor function; *Node* is a constructor for a tree; '_' denotes an anonymous variable that corresponds to a parent node; *lt* and *rt* are the left and right subtrees, respectively; and *Tip* is a leaf node. The symbols & and => denote conjunction and implication, respectively.

Consider data conversion between NAT and BTREE. Let f be a function that maps from NAT to BTREE and g be a function that maps from BTREE to NAT. We assume that these functions are defined inductively in the following form,

```
f: NAT --> BTREE
f(Suc(n)) = c1(f(n)).
f(0)      = Tip.
```

```
g: BTREE --> NAT
g(Node(_,lt,rt)) = c2(g(lt), g(rt)).
g(Tip)           = 0.
```

where $c1$ is a function from BTREE to BTREE, and $c2$ is a function from NAT× NAT to NAT.

Consider proving the following relationship between f and g,

$$g(f(n)) = n.$$

We use the following induction scheme IS on NAT to prove it.

$$\forall n.(\ (\forall n'.n' < n \Longrightarrow g(f(n')) = n') \Longrightarrow g(f(n)) = n\ ) \qquad \text{[IS]}$$

The proof proceeds by rewriting $g(f(n))$ in succession as follows:

$$\begin{aligned} g(f(n)) &= g(f(c2(n1, n2))) \\ &= g(Node(\_, f(n1), f(n2))) \\ &= c2(g(f(n1)), g(f(n2))) \\ &= c2(n1, n2) \end{aligned}$$

where $c2(n1, n2) = n$ for a certain $n1, n2 < n$.

The induction scheme IS is used to proceed from the third line to the fourth line. To succeed in this proof, there are two problems to be solved. The first is the progression from the first line to the second line: we have to prove the equality $f(c2(n1, n2)) = Node(\_, f(n1), f(n2))$. This problem is how to divide a natural number into two parts suitably. The second, and arguably more challenging problem, is that we cannot find $c1$ that defines f in a recursive form naturally. Consider the function f, which maps a natural number to a binary tree. The forms of the data structures $f(n)$ and $f(Suc(n))$ are considerably different, and the differences depend on the values of n. For example, compare the operation of adding a leaf node labeled "True" to the binary tree shown in Figure 1. In this figure, (a) shows the operation of adding a node to the binary tree that has two leaf nodes labeled "True," and (b) shows the operation of adding a node to the binary tree that has three leaf nodes labeled "True." These operations cannot be represented uniformly, therefore $f(n)$ cannot be uniformly defined for all n. Furthermore, even if f is defined, the proof over the properties on f is not straightforward.

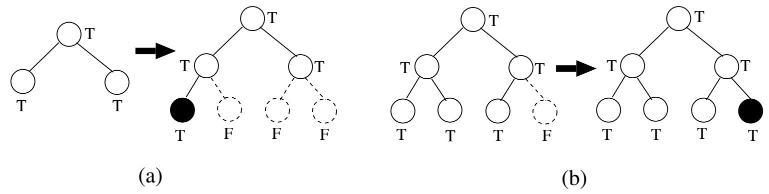

Figure 1: Operation on adding a "True" node to a binary tree

On the other hand, the definition of g is simpler because it is a conversion from a data type that is not totally-ordered to one that is totally-ordered.

2.2 Solution

To solve the problem shown above, we introduce the bit sequence BS as an intermediate data structure between NAT and BTREE.

2.2.1 Conversion from BS to BTREE

First, we describe the conversion from BS to BTREE. We choose a specific binary tree used in an electronic cash protocol. This tree is a complete binary tree and a Boolean value is attached to each node, the value of which is defined as a sum of the values of its left and right subtrees.

Let $b_0 b_1 \ldots b_n$ be a sequence of Boolean values that corresponds to a natural number k. Then $k = b'_0 \cdot 2^n + b'_1 \cdot 2^{n-1} + \ldots + b'_n \cdot 2^0$ where $b'_i = 1/0$ for $b_i = True/False$ $(1 \leq i \leq n)$, respectively. We encode $b'_0 \cdot 2^n$ as the left tree and the remainder as the right tree. The function from BS to BTREE is defined as follows:

```
bs_to_btree() = (Tip,false)
bs_to_btree(b#bs) = (if b
    then Node(true, create_btree(true,length(bs)), bs_to_btree(bs) )
    else Node(true, bs_to_btree(bs), create_btree(false,length(bs)) )
    )
```

where # is an operator combining the head and tail of the list; *Tip* indicates a leaf node; the function *create_btree*(*bool*, *nat*) creates a binary tree of height *nat* in which all nodes are labeled with *bool*; and *length*(*list*) denotes the length of *list* [1]. Intuitively, when scanning the data from the head of BS, descending by one bit corresponds to descending one subtree in BTREE. The induction scheme for the bit sequence is that if some property holds on a bit sequence *bs*, it also holds on *b#bs*. If $b = True$ (i.e., $k \geq 2^n$), then all nodes in the left subtree are labeled $True$; this is referred to as a $full\_tree$. However, if $b = False$ (i.e., $k < 2^n$), then none of the nodes in the right tree are labeled $True$; this is referred to as an *empty_tree*. Thus, the data types BS and BTREE are matched in their induction schemes.

Let a bit sequence be represented as a list, the elements of which are either $True$ or $False$. Figure 2 shows the conversion of the bit sequence $[True, False, True]$ into a binary tree. In this figure, the black trees are full trees, the white trees are empty trees, and the dotted trees are the others.

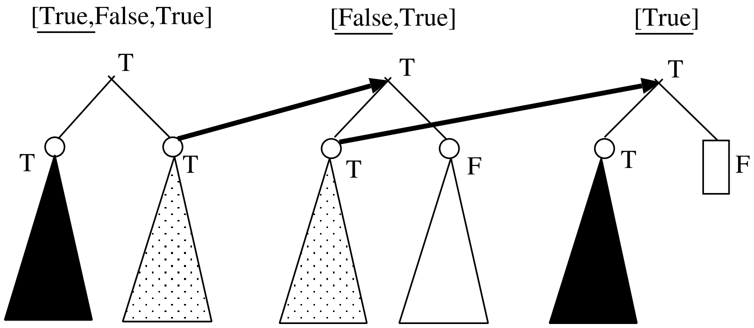

Figure 2: The mapping between a bit sequence and a binary tree

Because the binary tree corresponding to $[True, False, True]$ is not an empty tree, the root node is labeled $True$. Because the first bit in the sequence is $True$, the left tree is a full tree. The tree to the right of it is the tree corresponding to the remaining bit sequence $[False, True]$ after the first bit has been extracted. Now consider the coding of $[False, True]$; because the

[1] We can define a mirror tree in which left and right subtrees are exchanged.

first bit of the sequence is *False*, the right tree must be an empty tree, and the left tree is the tree corresponding to the remaining bit sequence. Then we consider the coding of [*True*]; because the first bit of the bit sequence is *True*, the left tree is a full tree, and the right tree is the binary tree corresponding to the remaining bit sequence. Finally, we obtain a *Tip* for [ ].

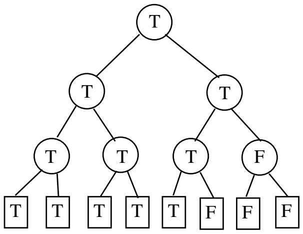

Figure 3: The binary tree for [*True*, *False*, *True*]

The binary tree obtained from [*True*, *False*, *True*] is thus as follows (Figure 3):

```
( True,
  ( True, ( True,True,True ),  ( True,True,True ) ),
  ( True, ( True,True,False ), ( False,False,False ) )
)
```

The function *bs_to_btree* can give an inductive definition that provides for the division of n into $n1$ and $n2$, corresponding to the left and right subtrees, respectively, in the proof process for IS in the previous subsection.

2.2.2 Conversion from NAT to BS

Next, we give a definition for the conversion from NAT to BS. We show two different models: one using a general function and one using a primitive recursive function.

First, we define the general function *naive_nat_to_bs*. This is defined in an iterative form and determines the value of a a bit sequence from the least-significant bit as follows:

```
naive_nat_to_bs(0) = [False]
naive_nat_to_bs(1) = [True]
naive_nat_to_bs(n) = (naive_nat_to_bs(div(n,2))) @ ([mod(n,2) = 1])
```

where @ is the concatenation operator for lists, $div(n, 2)$ and $mod(n, 2)$ return the result of division of n by 2 and its remainder, respectively.

Second, we define the primitive recursive function *nat_to_bs*. This is defined in a tail-recursive form and determines the value of a bit sequence from the most-significant bit. The definition is more complex than that of the iterative form.

```
nat_to_bs(0) = [false]
nat_to_bs(Suc(n)) = calc(lg(Suc(n)),Suc(n))

calc(0,m) = [true]
calc(Suc(n),m) = (if 2^n <= m) then True#(calc(n,m-2^(Suc(n))))
                         else False#(calc(n,m))

lg(n) = (if n<=1 then 0
             else Suc(lg (div(n,2))))
```

where *lg* is a key function that returns the place number of the bit sequence corresponding to a given argument, i.e., the natural number m that satisfies $m \leq \log\ n < m+1$. The following table shows the relationship of a natural number *nat*, a bit sequence *bs*, and the value of *lg*. Note that when n increases by 1, the location in the bit sequence, which is equal to the height of the tree, increases logarithmically.

nat	*bs*	*lg*
1	1	0
2	10	1
3	11	1
4	100	2
5	101	2
6	110	2
7	111	2
8	1000	3
9	1001	3
	⋮	

2.2.3 Conversion from NAT to BTREE

Finally, *nat_to_btree* is defined as the composition of the two functions *naive_nat_to_bs* and *bs_to_btree* or the composition of *nat_to_bs* and *bs_to_btree*. *nat_to_bs* and *bs_to_btree* are primitive recursive functions, whereas *naive_nat_to_bs* is a recursive function.

The definition of *nat_to_bs* is more suitable to an induction scheme than that of *naive_nat_to_bs*, because induction is typically applied from the head to the tail.

In the next two sections, we apply this formalization to the electronic cash protocol defined in [18] and prove its divisibility.

3 Modeling of an Electronic Cash Protocol

3.1 Ideal Electronic Cash

An electronic cash (e-cash) protocol is, in general, a combination of cryptography techniques, such as zero-knowledge proof and public key encryption. We consider the ideal e-cash protocol proposed by Okamoto [18]. In this protocol, a coin of some monetary value is encoded as a binary tree, and a payment function is defined over it. This binary tree approach makes e-cash efficient and unlinkable, and is used in many divisible e-cash schemes [7, 8, 15, 18]. We formalize this protocol and prove its divisibility on a data level, i.e., a user can spend a coin in several separate transactions by dividing its value without overspending if and only if a payment function satisfies the payment rules. This is one of the properties that an ideal e-cash protocol should satisfy. We define the data structure of *money* and define two primitive recursive functions of *money_amount* and *pay* on *money*. The definitions are shown using Coq code in this section.

3.2 Definition of *money*

A coin is represented as a binary tree called *money*. Each node of the tree represents a certain denomination. The root node is assigned the monetary value of the money, and the values of all other nodes are defined as half the value of their parent nodes.

Money is defined as an inductive function. It is a binary tree, where nodes are labeled with Boolean values. The label $True$ means that a node is usable, while $False$ means that it is not. For a given Boolean value b and a natural number n, we can create *money* in which all of the nodes have the label b.

```
Inductive money : Type :=
| Tip  : bool -> money
| Node : bool -> money -> money -> money.

Fixpoint create_money(b : bool)(n : nat) : money :=
 match n with
| 0 => Tip b
| S n’ => Node b (create_money b n’) (create_money b n’)
 end.
```

3.3 Creation of *money* from a natural number

cash is a function that creates *money* corresponding to a given natural number that is defined as a composition of *bs_to_money* and *naive_nat_to_bs* (or *nat_to_bs*). *bs_to_money* can be defined in a manner similar to that of *bs_to_btree*.

```
Definition cash(n : nat) : money := bs_to_money (naive_nat_to_bs n).
```

Note that *cash* is the money that satisfies a specific condition, whereas type *money* allows any complete binary tree whose nodes are labeled as Boolean values. For example, $Node(false, Tip(true), Tip(true))$ is an element of *money* but not a result of a function *cash*.

3.4 Calculation of the amount of *money*

The function *money_amount* computes the amount of money that can be used. If the root node is labeled $True$, the amount of the tree is the sum of that of the left tree and the right tree; otherwise, the amount is 0.

```
Fixpoint money_amount(m : money) : nat :=
 match m with
| Tip true => 1
| Tip false => 0
| Node true l r => money_amount l + money_amount r
| Node false _ _ => 0
 end.
```

3.5 Payment

Payment rules are set in [18] as follows: when we spend some amount from the coin, we search for a node (or combination of nodes) whose value is equal to the payment value, and then cancel these nodes; at the same time, all of the ancestors and all of the descendants are also canceled. Overspending is prevented if and only if these rules are satisfied.

The function *pay* corresponds to payment according to the payment rules. When we pay n from *money*, where n is less than or equal to the amount of *money*, then we pay all of n from the left tree, and the right tree remains as it is if the amount of the left tree is more than n.

Otherwise, we exhaust the left tree in the payment and pay the remainder from the right tree. For example, $money\_amount(pay(cash(13), 4) = 9$.

In the following code, eq_nat_dec and lt_dec are functions on NAT that denote '=' and '<', respectively. $change\_false$ is a function that changes the label of a node to $False$.

```
Fixpoint pay(m : money)(n : nat) : money :=
 match m with
| Tip true => Tip (if eq_nat_dec n 0 then true else false)
| Node true l r =>
if lt_dec (money_amount l) n
then Node true (change_false l) (pay r (n - money_amount l))
else Node true (pay l n) r
| _ => m
 end.
```

4 Proof of Properties

We prove three properties on the divisibility of the e-cash protocol. On proving these properties inductively, we apply the method proposed in Section 2.

In this section, we only show the outline of the proof, focusing on how induction is applied. The complete Coq proof is shown in the Appendix.

First, the distribution property over $money\_amount$ holds.

$$\begin{aligned} &money\_amount(Node(\_, left, right)) = \\ &\qquad money\_amount(left) + money\_amount(right) \quad \cdots (1) \end{aligned}$$

This can readily be proved. We rewrite the left side of the given equation.

4.1 Well-definedness of $cash$

The monetary value of $money$ created by $cash$ from a natural number is equal to that value. This property is represented as follows,

$$\forall\ n.\ (money\_amount\ (cash(n)) = n)$$

and is proved using the properties of the bit sequence. Below, we show a case in which the first bit b of the sequence is $True$. When it is $False$, the proof proceeds in the same manner.

$$\begin{aligned} &money\_amount(cash(n)) \\ &= money\_amount(bs\_to\_money(b\#bs)) && \cdots (2) \\ &= money\_amount(\ Node(true, \\ &\qquad\qquad bs\_to\_money(bs), create\_money(true, length(bs)\ ) && \cdots (3) \\ &= money\_amount(bs\_to\_money(bs)) + \\ &\qquad money\_amount(create\_money(true, (length(bs)))) && \cdots (4) \end{aligned}$$

Formula (2) is obtained by unfolding $cash$, where $nat\_to\_bs(n) = b\#bs$, formula (3) is obtained by unfolding bs_to_money, and formula (4) is obtained by the distribution property of $money\_amount$ (1).

The first term of formula (4) is transformed as follows.

$$
\begin{aligned}
&money\_amount(bs\_to\_money(bs)) \\
&= money\_amount(bs\_to\_money(nat\_to\_bs(n - 2^{lg\ n)})) \quad \cdots (5) \\
&= money\_amount(cash(n - 2^{lgn})) \quad \cdots (6)
\end{aligned}
$$

Formula (5) is obtained using case split tactics on n, property of lg and several other tactics. Formula (6) is obtained by unfolding $cash$.

The second term of formula (4) is transformed as follows:

$$
\begin{aligned}
&money\text{-}amount(create\_money(true, length(bs))) \\
&= money\_amount(cash(2^{lgn})) \quad \ldots (7)
\end{aligned}
$$

Here we use the following induction scheme of NAT.

$$
\begin{aligned}
&\text{if } \forall\ k;\ k < n,\ money\_amount(cash(k)) \ = \ k, \\
&\text{then } money\_mount(cash(n)) \ = \ n \text{ holds.}
\end{aligned}
$$

We can prove $2^{lg\ n} \leq n$ and $n - 2^{lg\ n} < n$. If $2^{lg\ n} < n$, we apply this type of induction to both formulas (6) and (7), and so formula (4) is finally transformed into the following form:

$$
\begin{aligned}
&money\_amount(cash(2^{lg\ n})) + money\_amount(cash(n - 2^{lg\ n})) \\
&= 2^{lg\ n} + (n - 2^{lg\ n}) \\
&= n
\end{aligned}
$$

In case $2^{lg\ n} = n$, the proof is simpler without using induction.

4.2 Well-definedness of *pay*

The amount remaining after payment is the difference between the original value and the payment value. This property is represented as follows [2]:

$$\forall\ n.\forall\ m.\ (money\_amount(pay(cash(n), m)) = n - m)$$

This is derived from the following lemma:

$$\forall\ c.\forall\ n.\ (money\text{-}amount(pay(c, n)) = money\text{-}amount(c) - n$$

which is proved as follows. When we pay n from c, if n does not exceed the amount of c, we pay $m1$ from the left subtree as far as possible, and pay the remainder $m2$ from the right tree. $m1$ and $m2$ are determined as follows: if $n < money\_amount(left)$, then $m1 = n$ and $m2 = 0$; otherwise, $m1 = money\_amount(left)$ and $m2 = n - money\_amount(left)$. This is proved using induction. Below, we show an inductive case, because the proof is simple for the base case.

$$
\begin{aligned}
&money\_amount(pay(c, m1 + m2)) \\
&= money\_amount(pay(Node(\_, left, right), m1 + m2)) \quad \cdots (8) \\
&= money\_amount(pay(left, m1)) + money\_amount(pay(right, m2)) \quad \cdots (9) \\
&= money\_amount(left) - m1 + money\_amount(right) \ - \ m2 \quad \cdots (10) \\
&= (money\_amount(left) \ + \ money\_amount(right)) \ - \ (m1 \ + \ m2) \\
&= money\_amount(Node(\_, left, right)) \ - \ (m1 + m2) \quad \cdots (11) \\
&= money\_amount(c) \ - \ (m1 + m2)
\end{aligned}
$$

[2] Note that $n - m = 0$ when $m > n$.

Formula (8) is obtained by expanding c. Formula (9) is obtained by the distribution property of *money_amount* and the property of payment on *money*. Formula (10) is obtained by applying induction to *money*. Formula (11) is obtained from the distribution property of *money_amount*.

4.3 Divisibility

Given that *ms* is a list of values, each of which corresponds to a transaction, if a user pays from the head of this list in succession, then the remainder is the correct value. This property is represented as follows:

$$\begin{aligned} &\forall n.\forall ms.(\ \ n \geq listsum(ms) \\ &\Longrightarrow money\_amount\ (foldl(pay(cash(n), ms)))\ =\ n\ -\ listsum(ms)\ ) \end{aligned}$$

Here, $foldl$ and $listsum$ are the functions handling a list. Let ms be a list $[m1, \ldots, mk]$. $foldl(pay(c, ms))$ is rewritten in the following form:

$$(pay(\ \ldots\ (pay(c, m1),\ \ldots\ mk)$$

and $listsum(ms)$ is rewritten in the following form:

$$m1\ +\ \ldots\ +\ mk$$

This theorem can be proved using the result of the proofs for the above two properties of well-definedness.

5 Discussion

Modeling the e-cash protocol and proving the properties described in the previous sections were performed using the theorem provers Isabelle/HOL and Coq, respectively. First, we compare these two provers.

Both of them are interactive theorem-proving environments based on inductive theorem proving. The data types and functions are defined in recursive form, and the proof proceeds by connecting suitable tactics.

Isabelle/HOL has a more powerful engine for automatic proof than Coq. A proof may succeed simply by using the 'auto' command without connecting multiple lemmas in Isabelle/HOL, whereas a user must specify tactics manually in Coq. However, the proof procedure in Coq is easier to understand.

Both provers are based on typed logics and adopt higher-order functions. In Coq, type-checking is richer and proof-checking is reduced to type checking. It requires the user to prove the termination of a recursive function. Isabelle/HOL also requires the user to prove the termination of a recursive function, but has a stronger automatic mechanism, and user does not need to supply much input.

We developed a model for an e-cash protocol, and set out to prove three properties of the model using Isabelle/HOL, and subsequently translated the model into Coq. The translation was basically straightforward. The definition of a primitive recursive function using *primrec* in Isabelle/HOL was translated to *Fixpoint* in Coq; the definition of a recursive function using fun was translated to *Function*; the definition of a non-recursive function using $definition$ was translated to $Definition$. In addition, we must prove the termination of the function introduced via *Function*.

The tactics used in the proofs of two provers were quite different. Generally, more tactics are required in Coq; however, the proof in Coq provides a useful basis from which to make a proof in Isabelle/HOL. Actually, we completed the proof of the unproved lemma in Isabelle/HOL by referring to the proof in Coq. From this experience, it is expected that similar modeling with other provers is possible and that the existing proofs will be useful as a basis for forming proofs in those other provers. All definitions and proofs in Isabelle/HOL and Coq are shown in [22].

In the field of protocol verification, there are a number of works on the verification of security protocols using Isabelle/HOL (e.g.,[2]). However, they mainly proved security or safety of protocols. To the best of our knowledge, there exists no research on a proof that focuses on the divisibility of electronic cash protocols using a theorem-proving approach.

Bijection between the set of natural numbers and rooted trees has been discussed in several works [5, 6, 9]. In these works, prime decomposition of the natural numbers was used to construct the corresponding rooted tree. They used an incomplete tree, in which each prime number forms an individual branch. It appears to be impossible to define this bijection in an inductive manner. Attempts to provide the mechanical proof were not made, and the correctness of the methods has not been proved using theorem provers. On the other hand, we propose a method for automated theorem proving using a complete binary tree.

The data conversion method described here is applicable to conversion from an inductively defined data structure with one successor to one with n successors by introducing a list in which each element takes n values as an intermediate data structure instead of a bit sequence. Following this transformation, the proof can proceed in a similar manner to the one shown in this paper, although the number of lemmas would increase. Moreover, it can be extended to data conversion from data structures with m successors to data structures with n successors by composing two data-conversion functions, i.e., one from an inductively defined data structure with m successors to that with one successor, and then a second function from an inductively defined data structure with one successor to one with n successors.

6 Conclusion

We have described a function for converting data from a natural number to a binary tree, and have given an inductive proof for its well-definedness. We have described a method of introducing a bit sequence as an intermediate data structure to provide a model for inductively defined data structures with different induction schemes. We formalized this method, developed a computational model based on it, and applied it to an electronic cash protocol. We succeeded in proving the properties of divisibility of the protocol using the theorem provers Isabelle/HOL and Coq, and discussed the effectiveness of the method.

In future, we would like to investigate other functions and/or properties on such a binary tree that is handled here, and develop methods of their inductive proof.

References

[1] Arsac, W., Bella, G., Chantry, X. and Compagna, L. : *Multi-Attacker Protocol Validation*, J. of Automated Reasoning 46(3-4):353-388 (2011).

[2] Bella, G., Massacci, B. and Paulson, L. : *Verifying the SET Purchase Protocols*, J. of Automated Reasoning 36:5 37 (2006)

[3] Bella, G. : *Inductive Verification of Smart Card Protocols*, J. of Computer Security 11(1):87-132 (2003).

[4] Bertot, Y. and Castŕan, P. : *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*, Springer Verlag (1998).

[5] Beyer, T. and Hedetniemi, S. M. : *Constant Time Generation of Rooted Trees*, SIAM J. Comput., 9(4):706-712 (1980).

[6] Cappello, P. : *A New Bijection between Natural Numbers and Rooted Trees*, 4th SIAM Conference on Discrete Mathematics (1988).

[7] Chan, A., Frankel, Y. and Tsiounins, Y. : *Easy Come - Easy Go Divisible Cash*, EUROCRYPT98, pp. 561-575 (1998).

[8] Canard, S. and Gouget, A. : *Divisible E-Cash Systems Can Be Truly Anonymous*, EUROCRYPT2007, pp.482-497 (2007).

[9] Göbel, F. : *On a 1-1-Correspondence between Rooted Trees and Natural Numbers*, J. of Combinatorial Theory, Series B(29):141-143 (1980).

[10] Hardin, D. S. (ed): *Design and Verification of Microprocessor Systems for High-Assurance Applications*, Springer Verlag (2010).

[11] Kaufmann,M., Monolios, P. and Moore, J. S. : *Computer-Aided Reasoning: An Approach*, Kluwer Academic Publishers (2000).

[12] Klein, G. et al. : *seL4: Formal Verification of an Operating-System Kernel*, Commun. ACM 53(6): 107-115 (2010).

[13] Kurita, T. and Nakatsugawa, Y. : *The Application of VDM to the Industrial Development of Firmware for a Smart Card IC Chip*, Int. J. of Software and Informatics 3(2-3):343-355 (2009).

[14] Leroy, X. *Formal Verification of a Realistic Compiler*, Communications of the ACM, 52(7):107-115 (2009).

[15] Nakanishi, T. and Sugiyama, Y. : *An Efficiently Improvement on an Unlinkable Divisible Electronic Cash System*, IEICE Trans. on Fundamentals, E85-A(19):2326-2335 (2002).

[16] Nipkow, T., Paulson, L. and Wenzel, M. : *Isabelle/HOL A Proof Assistant for Higher-Order Logic*, Springer Verlag (2002).

[17] Norell, U. : *Dependently Typed Programming in Agda*, Advanced Functional Programming 2008: pp.230-266 (2008).

[18] Okamoto, T. : *An Efficient Divisible Electronic Cash Scheme*, The proceedings of Crypto'95, pp.438-451 (1995).

[19] Owre,S., Rushby,J.M. and Shankar,N. : *PVS: A Prototype Verification System*, The proceedings of CADE-11, pp.748-752 (1992).

[20] Sprenger, C., Zurich, E. T. H., Basin, D., et al. : *Cryptographically Sound Theorem Proving* 19th IEEE Computer Security Foundations Workshop (2006).

[21] Takahashi, K. and Yoshimaru, S. : *Formalization of Data Conversion for Inductive Proof* Tunisia-Japan Workshop on Symbolic Computation in Software Science (SCSS 2009), pp.135-150 (2009).

[22] `http://ist.ksc.kwansei.ac.jp/~ktaka/EMONEY`

Appendix. Proof for E-cash protocol in Coq (tail-recursive version)

```
Require Import Omega List Arith Div2 Bool Recdef Wf_nat.

Theorem well_definedness_of_pay :
forall(m : money)(n : nat),
    n <= money_amount m
 -> money_amount(pay m n) = money_amount m - n.
Proof.
intros m.
induction m; intros n LNM; destruct b; simpl in *; try reflexivity.
 destruct (eq_nat_dec n 0).
  rewrite e; reflexivity.
  assert(n = 1) by omega.
  rewrite H; reflexivity.
 destruct (lt_dec (money_amount m1) n); simpl.
  rewrite IHm2; try omega.
  replace (money_amount (change_false m1)) with 0; try omega.
  destruct m1; simpl; reflexivity.
  rewrite IHm1; omega.
Qed.

Lemma bit_sequence_distribution :
forall(bs : list bool)(b : bool),
 money_amount(bs_to_money(bs ++ (b::nil))) =
        2 * money_amount(bs_to_money bs) + (if b then 1 else 0).
Proof.
intros bs b.
induction bs; simpl in *.
 destruct b; reflexivity.
 assert(forall(bl : bool),money_amount (create_money bl (length
        (bs ++ b :: nil))) =
  money_amount (create_money bl (length bs)) + money_amount
        (create_money bl (length bs))).
  intros; clear IHbs; destruct bl; induction bs; simpl; try omega.
 destruct a; simpl in *; rewrite IHbs; repeat rewrite plus_0_r;
        repeat rewrite plus_assoc; rewrite H; omega.
Qed.

Lemma one_reminder_div2 :
forall(n : nat),
 (2 * div2 n) + (if one_reminder n then 1 else 0) = n.
Proof.
intros n.
case_eq (one_reminder n); intros; induction n using lt_wf_ind.
destruct n; try discriminate.
destruct n; try reflexivity.
simpl in H.
simpl.
replace (S (div2 n + S (div2 n + 0) + 1))
with (S (S (2 * div2 n + 1))).
```

```
repeat f_equal.
rewrite H0; try reflexivity; try omega.
apply H.
simpl.
omega.
destruct n; try reflexivity.
destruct n; try discriminate.
simpl in H.
simpl.
replace (S (div2 n + S (div2 n + 0) + 0))
with (S (S (2 * div2 n + 0))).
repeat f_equal.
rewrite H0; try reflexivity; try omega.
apply H.
simpl.
omega.
Qed.

Theorem well_definedness_of_cash :
forall(n : nat),
 money_amount (cash n) = n.
Proof.
intros n.
induction n using lt_wf_ind.
destruct n; try (simpl; reflexivity).
destruct n; try (simpl; reflexivity).
unfold cash; rewrite bit_sequence_equation.
case_eq (one_reminder (S (S n))); intros;
 rewrite bit_sequence_distribution; unfold cash in H; rewrite H;
 try (apply lt_div2; apply lt_0_Sn);
 pose (one_reminder_div2 (S (S n)));
 rewrite H0 in e; apply e.
Qed.

Definition listsum(ns : list nat) : nat := fold_right plus 0 ns.

Definition payment_amount(m : money)(ns : list nat) : nat :=
        money_amount (fold_left pay ns m).

Theorem Divisibility.
forall(n : nat)(ns : list nat),
 listsum ns <= n ->
 payment_amount (cash n) ns = n - listsum ns.
Proof.
intros n ns LSM.
induction ns using rev_ind; simpl in *.
 rewrite <- minus_n_O.
 apply well_definedness_of_cash.
 unfold payment_amount in *.
 unfold listsum in *.
 rewrite fold_left_app;
 rewrite fold_right_app in *; simpl in *.
 rewrite plus_0_r in *.
 assert(fold_right plus 0 ns + x <= n).
  generalize n LSM.
  clear IHns LSM n.
  induction ns; intros; simpl in *.
   omega.
   assert(fold_right plus x ns <= n - a) by omega.
   apply IHns in H; omega.
 rewrite well_definedness_of_pay; rewrite IHns; simpl in *; try omega.
 replace (fold_right plus x ns) with ((fold_right plus 0 ns) + x).
 omega.
 clear IHns H LSM.
 induction ns; simpl.
  reflexivity.
  rewrite <- IHns; info omega.
Qed.
```