Formalization of Data Conversion for Inductive Proof

Kazuko TAKAHASHI
Kwansei Gakuin Uniersity
Sanada, Hyogo, JAPAN
ktaka@kwansei.ac.jp

Shizuo YOSHIMARU
Kwansei Gakuin Uniersity
Sanada, Hyogo, JAPAN
shizuo@kwansei.ac.jp

Abstract

We discuss a methodology for an inductive proof. Induction is a powerful technique used by many theorem provers, but its effectiveness is highly dependent on the user. It is especially difficult to apply effectively on a proof for a property over two data types that have different induction schemes. A natural number and a binary tree are fundamental data types and have different induction schemes. We propose the introduction of a bit sequence as intermediate data corresponding to these two data types so that inductive proof may be used. We formalize a method of data conversion between these two data types using an example of an ideal electronic cash protocol.

Keywords: Isabelle/HOL, electronic cash protocol, inductive method, divisibility

1 Introduction

Theorem proving is a technique for formal methods [15, 12, 2, 18]. There are many applications for theorem proving, from pure mathematics to practical hardware verification or security protocols, and many successful results have been reported. Inductive definitions of functions or data are frequently used in programming, especially for determining correctness, and are used by most theorem provers. Induction is a powerful method for proving a property on infinite data that is not used in a model checker [5], another technique for formal methods. However, most provers are simply environments that help users generate a proof by themselves, requiring that they navigate an "automatic prover" by providing appropriate lemmas. Investigating the methodology for the effective application of induction is thus very important.

When an inductive definition is given to a data type or a function, a prover generates an induction scheme based on it. Subgoals are generated using this scheme when an induction is used in the proof procedure. However, completing the proof is sometimes difficult when an induction is not used in the form expected by the user, because the induction scheme is not matched. Most properties proved in applications are those that hold for a single data type or a single induction scheme can be naturally applied even if a property holds for multiple data types. However, one sometimes must deal with a property that holds over data types with different induction schemes.

Consider data conversion from a natural number to a binary tree. In general, the following data types are inductively defined:

```
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 *lt* and *rt* are the left and right subtrees, respectively, and *node* is their parent node. Let NAT and BTREE be a natural number and a binary tree, respectively. NAT has a linear structure, and BTREE has a branching structure. When induction is applied to NAT, its goal is to show that if some property holds for n, then it also holds for $Suc(n)$. However, the objective for BTREE is to verify that if some property holds on *lt* and *rt*, then it also holds on $(node\ lt\ rt)$. To prove the correctness of a conversion from

NAT to BTREE, one must verify the property over two data types that have different induction schemes, resulting in a very difficult proof.

We originally encountered this problem while proving the correctness of an electronic cash (e-cash) protocol using a theorem prover [23]. Cash for some amount of money is represented in the form of a binary tree, and a payment function is defined on this tree. If one proves the correctness of this function, i.e., if one pays m from n, then the remainder is $n-m$, one must handle the two data types of NAT and BTREE which have different induction schemes.

Natural number, list, and binary tree are fundamental data types used in many applications. Several formalizations of the conversion from BTREE to NAT or LIST are provided. They use techiniques such as sorting the nodes of a tree into some order or embedding a tree structure in a non-flat list. However, formalizations for the inverse conversion are not provided and proofs for properties over these two data types have never been made. This conversion is essentially the generation of a branching structure from a linear structure according to a specific rule. An inductive definition is generally difficult.

The aim of this paper is to provide a methodology for converting from NAT to BTREE for the appropriate application of induction in proving a property over these data types. We introduce the bit sequence BS as an intermediate data representation between NAT and BTREE. We describe our approach using a proof of divisibility of e-cash using the Isabelle/HOL [15] theorem prover as a case study and discuss the formalization of data conversion from NAT to BTREE. Divisibility means that a user can spend an amount of money in several separate transactions by dividing its value without overspending. E-cash is implemented as a specific binary tree. We define the construction of BTREE from NAT, the calculation of the amount value of BTREE, and the payment function on BTREE. We prove the correctness of these functions and that divisibility holds. We also demonstrate the feasibility of our approach by illustrating how the proof proceeds.

This paper is organized as follows. In Section 2, we present in detail the problems of data conversion and show our solution. In Sections 3 and 4, we show a formalization and an inductive proof with Isabelle/HOL using the divisibility of e-cash as a case study. In Section 5, we present a discussion, and in Section 6, we present our conclusion.

2 Data Conversion Formalization

We first clarify the problem of proving a property over data types that have different induction schemes.

Let f be a function to convert NAT to BTREE and g be a function to convert BTREE to NAT.

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)) <- d1(g(lt), g(rt)).
g(Tip)          <- 0.
```

where $c1$ and $d1$ are some functions.

Consider proving a relationship between f and g,

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

The following subgoal is generated as an induction step:

$$\forall n.(\ g(f(n)) = n \Longrightarrow g(f(Suc(n))) = Suc(n)\ ).$$

The structural induction on NAT is used here, which shows

$$\forall n.P(n)$$

by proving

$$\forall n.(\ P(n) \Longrightarrow P(Suc(n))\ )$$

where P is a property.

Generally, the proof proceeds by rewriting $g(f(Suc(n)))$ in succession as follows:

$$\begin{aligned} g(f(Suc(n))) &= g(c1(f(n))) \\ &= c2(g(f(n))) \\ &= Suc(g(f(n))) \\ &= Suc(n). \end{aligned}$$

where $c2$ is some function. Induction hypothesis is used between the third and fourth lines.

For this proof to succeed, we must define functions $c1$ and $c2$ appropriately, but this is impossible. For one thing, there is no appropriate form of $c1$ in fact, because $f(Suc(n))$ cannot be defined using $f(n)$ inductively for an arbitrary n uniformly. Intuitively, the forms of trees $f(n)$ and $f(Suc(n))$ are largely different, and the differences depend on the values of n. For example, compare the operation of adding a leaf to an incomplete binary tree which has three leaf nodes, and the operation of adding a leaf to a complete binary tree which has four leaf nodes (Figure 1). These operations cannot be represented uniformly. In addition, there is no appropriate form of $c2$, because the region of f is BTREE, and $c1(f(n))$ should provide two arguments to g, although it actually provides only one. Even if these functions could be defined, the proof would not succeed because NAT has a linear data structure whereas BTREE has a branching data structure, and their induction schemes are not matched.

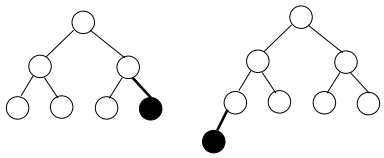

Figure 1: Operation on adding an node to a binary tree

We therefore provide another definition to f so that the result can provide two arguments to g, and use a different type of induction. We divide a natural number n into $n1$ and $n2$, which correspond to the left and right trees, respectively, of the data after the conversion. Then, the proof proceeds by rewriting $g(f(Suc(n)))$ as follows:

$$\begin{aligned} g(f(n)) &= g(f(n1+n2)) \\ &= g(node\ f(n1)\ f(n2)) \\ &= g(f(n1)) + g(f(n2)) \\ &= n1+n2 \end{aligned}$$

To succeed in rewiting the first line to the second line, we must define an appropriate function f satisfying the above requirement, and in rewiting the second line to the third line, we must define g satisfying the following property:

$$\forall lt.\forall rt.g(node\ lt\ rt) = g(lt) + g(rt).$$

To succeed in rewiting the third line to the fourth line, the complete induction for NAT is used here, which shows

$$\forall n.P(n)$$

by proving

$$\forall n.(\ \forall m.m < n;\ P(m) \Longrightarrow Pn)\ )$$

where P is a property.

The definition of g is easier because it is the conversion from the data type not in the total order to that in the total order. However, the definition of f in the inverse direction is very difficult. We introduce some intermediate data representation between NAT and BTREE to solve this problem.

There may be various BTREE specifications. Here, we choose the specific binary tree used in e-cash, a complete binary tree whose nodes are labeled (NAT,BOOL). Each node is assigned a value 2^n for some n, and the value of a node is equal to the sum of the values of its child nodes. Here n is the first argument of the label. Another argument is a boolean value that indicates whether the node is alive, and only live nodes can be used. There is also the constraint that if a node is not alive, then neither are any of its antecedents or descendants. We introduce a bit sequence BS as an intermediate data representation.

First we explain the conversion from BS to BTREE. Let $b_0b_1\ldots b_n$ be a bit sequence corresponding 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 $b_i = 0$ or $1 (1 \leq i \leq n)$. We encode $b_0 \cdot 2^n$ as a left tree and the remainder as a right tree. The function is defined as

```
bs-to-btree()     <- Tip
bs-to-btree(b#bs) <- (if b
    then (length(b#bs),False), full-tree(length(bs)), bs-to-btree(bs)))
    else (length(b#bs),False), bs-to-btree(bs), empty-tree(length(bs)))
    )
```

where # is an operator combining the head and the tail of a list.

Intuitively, scanning the data from the head of BS, going one bit lower corresponds to going one subtree lower in BTREE. The induction scheme for a bit sequence is that if some property holds on a bit sequence bs, it also holds on $b\#bs$. If $b_0 = 1$ (i.e., $k \geq 2^n$), then all nodes in the left subtree are alive; this is referred to as a $full\text{-}tree$. However, if $b_0 = 0$ (i.e., $k < 2^n$), then none of the nodes in the right tree are alive; this is referred to as an *empty-tree*, and the node passed is also a non-alive node. Thus, the data types BS and BTREE are naturally matched in their inductive schemes.

Let a bit sequence be represented as a form of a list whose element is $True$ or $False$. Figure 2 shows the conversion of a bit sequence $[True, False, True]$ to a binary tree. In this figure, a black tree indicates a full-tree, a white tree indicates an empty-tree, and a dotted tree indicates the others. First, because the binary tree corresponding to $[True, False, True]$ is not a full-tree, the root node is labeled $false$. Next, because the first bit of the bit sequence is $True$, the left tree is a full-tree, and the right tree is the tree corresponding to the remaining bit sequence $[False, True]$ after the first bit is extracted. Next we consider the coding of $[False, True]$. Because the first bit of the bit sequence is $False$, the right tree is an empty-tree, and the left tree is the tree corresponding to the remaining bit sequence. We then 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 an empty-tree for $[\,]$.

The BTREE obtained from BS [True,False,True] is thus as follows:

```
( (3,False),
  full-tree(2),
  ( (2,False),
```

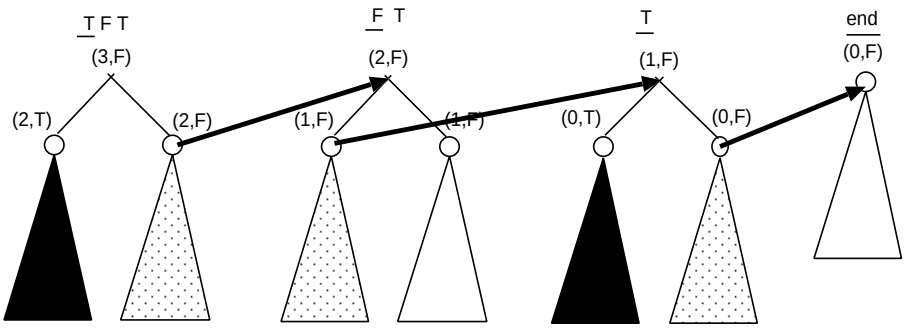

Figure 2: Correspondence between a bit sequence and a binary tree

```
      ( (1,False), full-tree(0), empty-tree(0) ),
      empty-tree(1)
)
```

The function *bs-to-btree* can provide a natural inductive definition that provides for the division of n into $n1$ and $n2$, corresponding to the left and right subtrees, respectively.

Next, we must provide a definition to the function that converts NAT to BS. When one converts a natural number to a bit sequence, he or she generally successively divides it by 2. The function *naiive-nat-to-bs* corresponding to this conversion is defined as a composite of the functions *n-to-b* and *rev*, where *n-to-b* is a function that determines the value of a bit sequence from a lower place and *rev* is a function that reverses the order of a sequence.

```
naiive-nat-to-bs(n) <- rev(n-to-b(n))

n-to-b(0) <- [0]
n-to-b(1) <- [0]
n-to-b(n) <- (n mod 2)#n-to-b(n div 2)

rev([])   <- [].
rev(x#xs) <- rev(xs)@[x].
```

where `@` is an operator of concatenation of lists.

The size of the bit sequence in this definition is not determined until an empty bit is encountered. However, the function *bs-to-btree*, which uses the result of *naiive-nat-to-bit* as an input, scans an input starting at the head of a list. This causes difficulty in proving the correctness of the conversion. To solve this problem, we introduce function *nat-to-bs* defined in tail-recursive form as equivalent to *naiive-nat-to-bs*. It determines the size of a bit sequence in advance and the value of a bit sequence from an upper place.

```
nat-to-bs(0) <- [0]
nat-to-bs(Suc(n)) <- calc(lg(Suc(n),Suc(n))

calc(0,m) <- [1]
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 (n div 2)))
```

The lg is a key function that returns a place number of the bit sequence corresponding to an 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 place number of the bit sequence, which equals 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
	⋮	

Using this conversion, we can prove the following properties inductively:

- The result of f satisfies the property required for BTREE.
- The composite of f and g is the identity function.
- The result of the function defined on BTREE is converted to the result of the corresponding function defined on NAT.

In the next two sections, we show the formalization of an e-cash protocol and prove its divisibility in Isabelle/HOL as an application of the formalization shown in this section.

3 Formalization of Electronic Money

3.1 Isabelle/HOL

Isabelle/HOL is a theorem-proving environment that is part of the HOL system [15]. Data types and functions are defined in recursive form, and the proof starts in the reverse direction. In other words, the prover first tries to verify the goal shown by the user; if it cannot be proved directly, the prover generates subgoals using tactics such as resolution and assumption. The prover repeats this process for each subgoal, and if all the subgoals are proved to be true, the verification terminates.

The program code shown in this section is that of Isabelle/HOL.

3.2 Ideal Electronic Money

An e-cash protocol is a combination of cryptography techniques such as zero-knowledge proof and public key encryption [7, 8, 11]. Okamoto identified the following six properties that should be satisfied by ideal e-cash protocols with no common specification or formalized properties: independence, security, untraceability, offline operation, transferability, and divisibility [17]. These properties have become the standard for ideal e-cash protocols [10, 13, 22, 24]. We consider an ideal e-cash protocol proposed by Okamoto. In this protocol, a coin of some monetary value is encoded as a kind of binary tree, and a payment function is defined on it. This binary tree approach makes e-cash efficient and unlinkable and is used in many divisible e-cash schemes [9, 10, 14, 16, 17]. The correctness of this approach is generally determined on the basis of cryptography, and there has been no previous work based on theorem proving. 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.

3.3 Money and Payment Rules

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

When we spend some amount from the coin, we search for a node(s) whose value(s) equals the payment value, and we cancel the node(s). At the same time, all of the ancestors and all of the descendants are canceled. Overspending is prevented if and only if these rules are satisfied.

[**rules of payment]**

1. When a node is canceled, all of its ancestor and descendant nodes are also canceled.
2. Each node can be canceled only once.

3.4 Data Conversion between NAT and BTREE

We define functions *cash* to create money for some natural number and *money-amount* to calculate the amount of money. The former is the conversion from NAT to BTREE, and the latter is its inverse.

3.4.1 Data Conversion from NAT to BTREE

If a natural number k is in the form of 2^n for some natural number n, then we say that k is a *full value*.

Then *money* is defined as a complete binary tree whose node is labeled by the pair *nat* and *bool*. A node labeled with $(n, true)$ means that the value of the node is 2^n and it can be used, whereas $(n, false)$ means that the value of the node is 2^n and it has been canceled. The former is called a *true node*, and the latter is called a *false node*.

Figure 3 is a coin corresponding to five cents.

types $money = (nat * bool)\ tree$

Now, *money* whose nodes are all $true$ is referred to as *full money*, and if all its nodes are $false$, it is referred to as *empty money*.

primrec
full-money :: $nat \Rightarrow money$
where

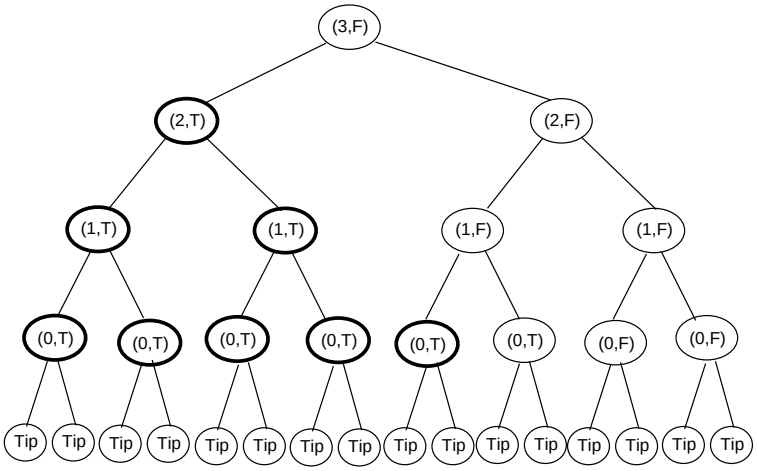

Figure 3: A coin corresponding to five cents (Cash 5)

full-money 0 = *Node* (*0*, *True*) *Tip Tip* |
full-money (*Suc n*) = *Node* (*Suc n*, *True*) (*full-money n*) (*full-money n*)

primrec
empty-money :: *nat* ⇒ *money*
where
empty-money 0 = *Node* (*0*, *False*) *Tip Tip* |
empty-money (*Suc n*) = *Node* (*Suc n*, *False*) (*empty-money n*) (*empty-money n*)

The *money* that satisfies the following conditions is called *valid money*:

1. It is a complete binary tree.
2. For each false node, all of its ancestor nodes and all of its descendant nodes are false nodes.

In the original definition of the binary tree approach, a binary tree of *money* may be incomplete and the values for both children of a node may not be equivalent, as far as it obeys the condition that the value of each node is the sum of its children nodes. Moreover, a set of nodes used for payment is arbitrary as far as it obeys the rules of payment. For simplicity, in this paper we use a complete binary tree, i.e., one for which the size of the left and right subtrees is the same and whose node is assigned a value of 2^i for $i = 0, 1, \ldots$. In addition, when we construct a tree for a given value, a label for each node is given in a depth-first manner, and when we pay an amount of money, we use nodes from top to bottom and from letf to right sequentially. These rules guarantee the uniqueness of the structure of *money* for each coin.

The function *cash* is defined using a bit sequence so that a suitable induction scheme may be applied. It creates *money* from a given natural number. If the number is a full value, full money is created. Otherwise, *money* is created using *bs-to-money* and *nat-to-bs*. *bs-to-money* is almost the same as *bs-to-btree* shown in Section 2.

primrec
cash :: *nat* ⇒ *money*
where
cash-base: *cash 0* = *Tip* |
cash-step: *cash* (*Suc n*) = (*if* (*Suc n*) = *2* ^ *lg* (*Suc n*)

then full-money (*lg* (*Suc n*))
else bs-to-money (*nat-to-bs* (*Suc n*)))

For example, $cash\ 5 =$ $(Node\ (3, False)\ (full\text{-}money\ 2)\ (Node\ (2, False)$
$(Node\ (1, False)\ (full\text{-}money\ 0)\ (empty\text{-}money\ 0))\ (empty\text{-}money\ 1)))$

3.4.2 Data Conversion from BTREE to NAT

The function *money-amount* computes the amount of money that can be used. If a given natural number is a full value, *money* is full money. Otherwise, if the root node of *money* is a false node, the amount of the left tree is 2^{n-1}, and the amount of the right tree is $n - 2^{n-1}$.

primrec
money-amount :: *money* ⇒ *nat*
where
money-amount Tip = *0* |
money-amount (*Node x l r*) = (*if usable x*
then 2 ^ amount x
else money-amount l + *money-amount r*)

For example, *money-amount*$(cash\ 5) = 5$

Moreover, we add the following axiom on bit sequences that relates the bit sequence to the corresponding *money*, which can be proved.

axioms
money-amount-for-bit-seq: *bit-seq n* = *b*#*bs*
⟹ *money-amount* (*bs-to-money bs*) = *money-amount* (*bs-to-money* (*bit-seq* (*n* − *2 ^ lg n*)))

3.5 Functions on BTREE

First, we define the function corresponding to the property that *money* should satisfy.

Generally, *money* is an arbitrary binary tree whose nodes are labeled with the pair *nat* and *bool*. The data that are valid as money should satisfy two conditions: they form a complete binary tree, and all ancestor nodes and all descendant nodes of a false node are false nodes.

primrec
is-complete-bt :: *money* ⇒ *bool*
where
is-complete-bt Tip = *True* |
is-complete-bt (*Node x l r*) = (*if amount x* = *0*
then l = *Tip* ∧ *r* = *Tip*
else if l = *Tip* ∨ *r* = *Tip*
then False
else (*amount x* = *Suc* (*money-capacity l*)) ∧ (*money-capacity l* = *money-capacity r*) ∧ *is-complete-bt l* ∧
is-complete-bt r)

primrec
satify-payment-rule :: *money* ⇒ *bool*
where
satisfy-payment-rule Tip = *True* |

```
satisfy-payment-rule (Node x l r) = (if l = Tip ∨ r = Tip
  then True
  else if usable x
    then is-full-money l ∧ is-full-money r
    else ¬ (is-full-money l ∧ is-full-money r) ∧ (satisfy-payment-rule l ∧ satisfy-payment-rule r)
```

```
definition
  is-valid-money :: money ⇒ bool
where
  is-valid-money c ≡ is-complete-bt  c∧ satisfy-payment-rule c
declare check-money-def [simp]
```

Next, we define the payment function on BTREE.

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 right tree, and the left tree remains as it is if the amount of the right tree is more than n. Otherwise, we exhaust the right tree in the payment and pay the remainder from the left tree.

```
primrec
  pay :: money ⇒ nat ⇒ money
where
  pay Tip n = Tip |
  pay (Node x l r) n = (if n = 0
    then Node x l r
    else if (usable x ∧ 2 ^ amount x < n)
      then empty-money (amount x)
      else if (money-amount r < n)
        then Node (amount x, False) (pay l (n − money-amount r)) (empty-money (money-capacity r))
        else Node (amount x, False) l (pay r n))
```

For example, $money\text{-}amount(pay\ (cash\ 13)\ 9) = 4$

In the above definition, $(usable\ x)$ means that the node x is a true node, $(amount\ x)$ indicates the labeled number of the node x, and $(money\text{-}capacity\ r)$ indicates the labeled number of the root node of tree r.

4 Proof

4.1 Basic Concept

We prove four important properties that hold over two data types NAT and BTREE.

1. **validity**
   $\forall n.is\text{-}valid\text{-}money(cash\ n)$

   $\forall c.\forall ms.\ is\text{-}valid\text{-}money(foldl\ pay\ c\ ms)$

2. **the relationship of** *cash* **and** *money-amount*
   $\forall\ n.\ (money\text{-}amount\ (cash\ n) = n)$

3. **well-definedness of** *pay*
   $\forall\ n.\forall\ m.\ (money\text{-}amount\ (pay\ (cash\ n)\ m) = n - m)$

4. **divisibility**
   $\forall n.\forall ms.(\ n \geq listsum\ ms \Longrightarrow money\_amount\ (foldl\ pay\ (cash\ n)\ ms)\ =\ n\ -\ listsum\ ms\ )$

Here, $foldl$ and $listsum$ are the functions handling a list that are defined in the Isabelle/HOL library. 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:

$$(m1\ +\ \ldots\ +\ mk)$$

We show only the outline of the proof. The main proof code in Isabelle/HOL is shown in Appendix. The proof includes about 60 lemmas in total.

4.2 Validity

The first specification indicates that the entity generated by *cash* is a valid money. The second specification indicates that the remainder of any transaction is a valid money. Both specifications can be easily proved.

4.3 Relationship of *cash* and *money-amount*

The specification indicates that the amount of *money* created by *cash* from a natural number is equal to that value. This property is proved using the properties of a bit sequence. The proof is conducted depending on the form of natural number n.

(o) distribution property of *money-amount*
The function *money-amount* satisfies the following property of distribution over *money*.

$$\begin{aligned} &money\text{-}amount(Node\ x\ left\ right)\ = \\ &\qquad money\text{-}amount(left) + money\text{-}amount(right). \quad \cdots(1) \end{aligned}$$

It can be proved easily.

(i) n is a full value

$$\begin{aligned} &money\text{-}amount(cash\ 2^n) \\ &= money\text{-}amount(full\text{-}money\ (lg\ 2^n))) && \cdots(2) \\ &= money\text{-}amount(full\text{-}money\ n) && \cdots(3) \\ &= money\text{-}amount(Node\ (n,True)\ (full\text{-}money\ n-1)\ (full\text{-}money\ n-1)) && \cdots(4) \\ &= 2^n && \ldots(5) \end{aligned}$$

Formula (2) is obtained by unfolding *cash*, and formula (3) is obtained by the property of lg. Formula (4) is obtained by unfolding $full$-*money*, and formula (5) is obtained by unfolding *money-amount*.

(ii) n is not a full value
In this case, the root node of *money* for $(cash\ n)$ is a false node, and the left tree is full money.

$$
\begin{aligned}
&\textit{money-amount}(\textit{cash}\ n)\\
&= \textit{money-amount}(\textit{bs-to-money}\ (b\#bs)) && \cdots(6)\\
&= \textit{money-amount}(\textit{Node}\ ((\textit{length}\ b\#bs)\ \textit{False})\\
&\qquad\qquad (\textit{full-money}\ (\textit{length}\ bs))\ (\textit{bs-to-money}\ bs)) && \cdots(7)\\
&= \textit{money-amount}(\textit{full-money}\ (\textit{length}\ bs)) +\\
&\quad \textit{money-amount}(\textit{bs-to-money}\ bs) && \cdots(8)
\end{aligned}
$$

Formula (6) is obtained by unfolding *cash*, where $\textit{nat-to-bs}\ n = b\#bs$. Formula (7) is obtained by unfolding *bs-to-money*, and formula (8) is obtained by the distribution property of *money-amount* (1).

In this case, because $\textit{nat-to-bs}\ n\ =\ b\#bs for\ n > 0$ holds, the following also holds.

$$
\begin{aligned}
&\textit{cash}\ 2^{lg\ n}\\
&= \textit{full-money}(lg\ 2^{lg\ n}) && \ldots(9)\\
&= \textit{full-money}(lg\ n) && \ldots(10)\\
&= \textit{full-money}(\textit{length}\ bs) && \ldots(11)
\end{aligned}
$$

Formula (9) is obtained by unfolding *cash* and from the property of lg. Formula (10) is obtained from the property of lg, and formula (11) is obtained from the property of *bit-sequence*.

Therefore, the first term of formula (8) is transformed as follows.

$$
\begin{aligned}
&\textit{money-amount}(\textit{full-money}\ (\textit{length}\ bs))\\
&= \textit{money-amount}(\textit{cash}\ 2^{lgn}) && \ldots(12)
\end{aligned}
$$

The second term of formula (8) is transformed as follows.

$$
\begin{aligned}
&\textit{money-amount}(\textit{bs-to-money}\ bs)\\
&= \textit{money-amount}(\textit{bs-to-money}\ (\textit{nat-to-bs}\ (n-2^{lg\ n)})) && \cdots(13)\\
&= \textit{money-amount}(\textit{cash}\ n-2^{lgn}) && \cdots(14)
\end{aligned}
$$

Formula (13) is obtained by the axiom $\textit{money-amount-for-bit-seq}$, where $\textit{nat-to-bs}\ n = b\#bs$. Formula (14) is obtained by unfolding *cash*.

Here we use the complete induction on NAT named $\textit{nat-less-induct}$.

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

Because $2^{lg\ n} < n$ and $n - 2^{lg\ n} < n$, we apply this type of induction to both formulas (12) and (14), and formula (8) is finally transformed to the following form.

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

4.4 Well-definedness of *pay*

The specification indicates that the amount remaining after payment is the difference between the original value and the payment value (Figure 4).

This is derived by proving the following lemma.

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

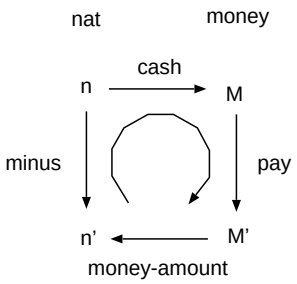

Figure 4: Well definedness of *pay*

The lemma is proved as follows. When we pay n from *money* c, if n does not exceed the amount of c, we pay $m2$ from the right subtree as far as possible and pay the remainder $m1$ from the left tree. The $m1$ and $m2$ are determined as follows: if $n < (\textit{money-amount right})$, then $m1 = 0$ and $m2 = n$; otherwise, $m1 = n - (\textit{money-amount right})$ and $m2 = (\textit{money-amount right})$.

$$
\begin{aligned}
&\textit{money-amount}(\textit{pay } c\ (m1+m2)) \\
&= \textit{money-amount}(\textit{pay } (\textit{Node } x\ \textit{left right})\ (m1+m2)) && \cdots(15)\\
&= \textit{money-amount}(\textit{pay left } m1) + \textit{money-amount}(\textit{pay right } m2) && \cdots(16)\\
&= (\textit{money-amount}(\textit{left}) - m1) + (\textit{money-amount}(\textit{right}) - m2) && \cdots(17)\\
&= (\textit{money-amount}(\textit{left}) + \textit{money-amount}(\textit{right})) - (m1 + m2) \\
&= \textit{money-amount}(\textit{Node } x\ \textit{left right}) - (m1+m2) \\
&= \textit{money-amount}(c) - (m1+m2)
\end{aligned}
$$

Formula (15) is obtained by expanding c. Formulas (16) and (18) are obtained by the distribution property of *money-amount* (1). Formula (17) is obtained by applying induction on *money*.

4.5 Divisibility

Given that *ms* is a list of values, each of which corresponds to a transaction, this specification indicates that if a user pays from the head of this list in succession, then the remainder is the correct value. This is a main theorem and it is easily proved using the properties 1,2 and 3.

5 Discussion

Many studies have been conducted of the proof of security protocols using Isabelle/HOL. Paulson succeeded in verifying the SET protocols [21], and Bella showed the verification of smart cards [3]. Many applications for proving complicated security protocols such as Kerberos [4] and TLS [20] were presented later. In Paulson's approach, a protocol is modeled as a sequence of events, and the sequences that may occur are modeled as a set [19]. If one event occurs, one sequence of the events is also added. Therefore, inductive definition is natural, and inductive proof on this data type can also be applied naturally.

These applications treat the protocol level and there has been little work at the data level. At the data level, one must treat the property over different data types. And when one treats the property over NAT and BTREE, which have different induction schemes, one must arrange things so that induction can lead to a successful proof. We have solved this problem by introducing an intermediate data type. A bit sequence was used as this intermediate data type because its properties resemble those of the

BTREE used in e-cash. In general, the data type used as intermediate data depends on the specification of BTREE. In addition, we have proposed an algorithm for converting NAT to BS that determines the element from the head of a resulting bit sequence.

The most important point is that we have provided a definition for the conversion function in recursive form so that a natural number can be divided into two parts corresponding to the left and right subtrees. We recognize the effectiveness of this approach in the proof of the relationship of *cash* and *money-amount* and that of the well-definedness of *pay*.

6 Conclusion

The method for converting data from NAT to BTREE that we have proposed makes possible an inductive proof for the properties over data with different induction schemes. For instance, Because many theorem provers use induction as a proof scheme, and data types such as NAT, LIST, and BTREE are frequently used, our approach is effective for use in many other applications. We have used a bit sequence as an intermediate data type to successfully apply induction to a proof and have illustrated this approach using an e-cash protocol.

We are considering the application of our approach to other problems. For example, Blanchette recently conducted an interesting study that formalized the textbook implementation of the Huffman algorithm in Isabelle/HOL and demonstrated the correctness of the algorithm [6]. The Huffman algorithm involves coding on binary trees and is used mainly for data compression. It would be interesting to apply our approach to that application, because our work can be considered a proof related to the coding of a natural number on a binary tree.

References

[1] Abadi,M. and Rogaway,P.: *Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption)*, J. Cryptology 20(3): p.395 (2007).

[2] Barthe,G., Forest,J., Pichardie,D. and Rusu,V. : *Defining and Reasoning about Recursive Functions: A Practical Tool for the Coq Proof Assiatant*, FLOPS2006, (2006).

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

[4] Bella,G. and L.Paulson: *Kerberos Version IV: Inductive Analysis of the Secrecy Goals*. 5th European symposium ob Research in Computer Security (ESORICS 98), pp.361-375 (1998).

[5] Bérard,B., M.Bidoit, A.Finkel, F.Laroussibie, A.Petrucci, Ph.Schnoebelen and P.Mckenzie: *Systems and Software Verification - Model-Checking Techniques and Tools* - . Springer (1999).

[6] Blanchette,J.C. : *Proof Pearl: Mechanizing the textbook proof of Huffman's algorithm in Isabelle/HOL*. J. of Automated Reasoning, 43(1), pp.1-18 (2009).

[7] Brands, S. : *Off-Line Electronic Cash Based on Secret-Key Certificates*, Proceedings of the Second International Symposium of Latin American Theoretical Informatics (LATIN '95), Valparaiso, Chili, (1995).

[8] Chaum, D., Fiat, A. and Naor, M. : *Untraceable Electronic Cash*, Advances in Cryptology - CRYPTO '88 Proceedings, Springer Verlag, pp.319-327, (1990).

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

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

[11] Feng, B. : *Colluding Attacks to a Payment Protocol and Two Signature Exchange Schemes*, Advances in Cryptology - ASIACRYPT, Springer Berlin / Heidelberg, Vol.3329/2004, pp.417-429 (2004).

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

[13] Liu,J.L., Wei, V.K. and Wong, S.H. : *Recoverable and untraceable E-cash*, EUROCON'2001, Trends in Communications, International Conference on, pp.132-135, (2001).

[14] Nakanishi,T. and Sugiyama,Y.: *An Efficiently Improvement on an Unlinkable Divisible Electronic Cash System*. IEICE Trans. on Fundamentals, Vol.E85-A, No.19, pp.2326-2335 (2002).

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

[16] Okamoto, T. and Ohta, K. : *Disposable Zero-Knowledge Authentications and Their Applications to Untraceable Electronic Cash*, Proceedings of Crypto 89, pp.481-496 (1990).

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

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

[19] Paulson, L. : *The Inductive Approach to Verifying Cryptographic Protocols*, J. of Computer Security, Vol.6, pp.85-128 (1998).

[20] Paulson, L. : *The Inductive Analysis of the Internet Protocol TLS*, ACM Transactions on Computer and System Security, pp.332-351 (1998).

[21] Paulson, L. : *Making Sense of Specifications: The Formalization of SET (Transcript of Discussion)* , Security Protocols Workshop 2000, pp.82-86 (2000).

[22] Schneier, B. : *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, John Wiley & Sons Inc, ISBN 978-0471117094,(1995).

[23] Shizuo, Y. and Takahashi, K. : *Formalization and Inductive Proof for Divisibility of an Electronic Cash*, IPSJ SIG Technical Report PRO-74, (2009), (In Japanese).

[24] Zhong,M. Feng,Y. and Yang,Y. : *Single-term divisible electronic cash based on bit commitment* Proceedings of Fifth IEEE Symposium on Computers and Communications (ISCC 2000). pp.280-285 (2000).

A The Proof Code in Isabelle/HOL for Divisibility of E-Cash

lemma *validity*: *is-valid-money* (*cash n*)
apply (*case-tac n*, *simp*)
apply (*simp only*:*cash-step*)
apply (*split split-if*, *simp*)
done

lemma *validity-for-list*: ⟦*is-valid-money c*; *money-amount c* ≥ *listsum ms*⟧ ⟹ *is-valid-money* (*foldl pay c ms*)
by (*simp only*:*check-money-amount-for-fold-pay*)

lemma *the-relation-of-cash-and-money-amount*: *is-valid-money* (*cash n*) ⟹ *money-amount* (*cash n*) = *n*
apply (*induct n rule*:*nat-less-induct*)
apply (*case-tac n*, *simp*)
apply (*case-tac n* = *2* ^ *lg n*)
apply (*simp only*:*money-amount-for-cash-with-complete-n*)
apply (*case-tac bit-seq n*)
apply (*simp only*:*bs-step-noteq-Nil*)
apply (*subgoal-tac money-amount* (*cash n*) = *money-amount* (*Node* (*length* (*a* # *list*), *False*) (*full-money* (*length list*)) (*bs-to-money list*)))
prefer *2*
apply (*rule develop-cash-in-money-amount*, *assumption*+)
apply (*subgoal-tac money-amount* (*Node* (*length* (*a* # *list*), *False*) (*full-money* (*length list*)) (*bs-to-money list*))

$= n$)
apply (*simp* (*no-asm-use*))
apply (*simp* (*no-asm*))
apply (*subgoal-tac money-amount* (*cash* (2 ^ *lg n*)) = *money-amount* (*full-money* (*length list*)))
prefer *2*
apply (*rule develop-cash-for-two-to-lg-in-money-amount*, *assumption*+)
apply (*subgoal-tac money-amount* (*cash* (2 ^ *lg n*)) + *money-amount* (*bs-to-money list*) = *n*)
apply (*simp* (*no-asm-use*))
apply (*subgoal-tac money-amount* (*cash* (*n* − 2 ^ *lg n*)) = *money-amount* (*bs-to-money list*))
prefer *2*
apply (*rule develop-cash-for-except-two-to-lg-in-money-amount*, *assumption*+)
apply (*subgoal-tac money-amount* (*cash* (2 ^ *lg n*)) + *money-amount* (*cash* (*n* − 2 ^ *lg n*)) = *n*)
apply (*simp only*:*hd-calc-bit-seq-True*)
apply (*subgoal-tac money-amount* (*cash* (2 ^ *lg n*)) = 2 ^ *lg n*)
prefer *2*
apply (*rule money-amount-of-left-induct*, *assumption*+)
apply (*subgoal-tac* 2 ^ *lg n* + *money-amount* (*cash* (*n* − 2 ^ *lg n*)) = *n*)
apply (*simp only*:*hd-calc-bit-seq-True*)
apply (*subgoal-tac money-amount* (*cash* (*n* − 2 ^ *lg n*)) = *n* − 2 ^ *lg n*)
prefer *2*
apply (*rule money-amount-of-right-induct*, *assumption*+)
apply (*subgoal-tac* (2 ^ *lg n*) + (*n* − 2 ^ *lg n*) = *n*)
apply (*simp only*:*hd-calc-bit-seq-True*)
apply (*simp only*:*associative-for-two-to-lg*)
done

lemma *well-defineness-of-pay-lemma*: ⋀*n*. ⟦*is-valid-money c*; *n* ≤ *money-amount c*⟧ ⟹ *money-amount* (*pay c n*)
= (*money-amount c*)− *n*
apply (*induct c*, *simp*)
apply (*case-tac n* = *0*, *simp*)
apply (*case-tac amount a* = *0*, *simp*)
apply (*case-tac c1* = *Tip* ∨ *c2* = *Tip*, *simp*)
apply (*case-tac* ¬ *usable a*, *simp*)
apply (*simp add*:*money-amount-on-check-full*)
done

lemma *well-defineness-of-pay*: *n* ≥ *m* ⟹ *money-amount* (*pay* (*cash n*) *m*) = *n* − *m*
by (*simp only*:*check-money-for-cash*
well-defineness-of-pay-lemma
money-amount-of-cash)

theorem *divisibility*: *n* ≥ *listsum ms* ⟹ *money-amount* (*foldl pay* (*cash n*) *ms*) = *n* − *listsum ms*
apply (*induct ms rule*:*rev-induct*, *simp-all*)
apply (*simp only*:*money-amount-of-cash*)
apply (*simp only*:*check-money-for-cash succeed-payment money-amount-of-cash*
check-money-for-fold-pay)
done