ISBN 0-612-10718-3

[illegible]

Edward Ho
80 Lansbury Drive
Scarborough, Ontario
Canada M1V 3H6

Date: December 11, 1995.

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Implementing Binary Trees in MIZAR-C** submitted by Edward Ho in partial fulfillment of the requirements for the degree of Master of Science.

[illegible]

Dr. H.J. Hoover

[illegible]

Dr. P. Rudnicki

[illegible]

Dr. B. Cockburn

Date: December 8, 1995.

Abstract

In programming via constructive proofs, the specifications of the program are first proven to be satisfiable and the code is then extracted from the proof. In this research, the feasibility of this programming methodology is being investigated by using and extending the MIZAR-C system, a natural deduction proof environment that extracts Lisp code from constructive proofs.

The MIZAR-C system is extended with a basic data type of bit strings. My main project was to implement a binary tree data structure in the system. In this thesis, we used definition by implementation to define binary trees. This approach defines recursive structures directly and gives an iterative test for correctness of construction. In addition, we defined a new inference rule which allows the introduction of a recursive function from a proof of its specifications.

Acknowledgements

Thanks to the Lord Almighty for giving me the wisdom to complete this degree.

Special thanks to my supervisor, Dr. H. J. Hoover for providing me with such an excellent research atmosphere, giving me invaluable advice on my program and constructive comments on this thesis, and letting me go to work before I finished all my writing. I also want to extend my appreciation to him and the Department of Computing Science for providing me with funding during my stay.

Thanks also to:

Dr. P. Rudnicki and Dr. B. Cockburn for their thorough review,

my parents for their patience and understanding,

Queenie for all her prayers and love.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Formal Methods

Formal methods in computing science are mathematical modelling techniques for describing properties of computer systems. These methods have a sound mathematical basis that provides the means for proving that the system design and implementation satisfy precisely specified properties. For a general introduction to formal methods, see [24].

Why are formal methods necessary? Honestly, the use of formal methods is not applicable to all systems due to the extra time, and thus cost, required to perform the formal proofs of specifications. However, there are systems where the cost of failure is extremely high. The extra cost of using formal methods is thus justifiable for these systems. There are also systems where no rigorous testing can test all possible actions. As a result, the use of formal methods is necessary to guarantee the correctness of implementation.

Although uncommon, formal methods have been applied to industrial projects. For example, IBM's CICS [15], a large, 20-year-old transaction-processing system, was redesigned using formal methods. It contains more than a half million lines of code. Commercial operations such as Praxis [6] are dedicated users of formal methods. They write specifications and develop software from them.

The use of formal methods is not an all-or-nothing approach. A useful strategy is to apply formal methods to the most critical portions of a system only. In this way, we can increase the reliability of the system with a justifiable cost. As formal methods become more common and well developed, we can formally describe and verify the correctness of more portions of a large complex system.

1.2 Programming via Constructive Proofs

One of the major research areas in formal methods is the use of constructive proofs to specify and produce correct programs. In traditional programming methodolgy, a program is specified, implemented, and then tested to satisfy the specifications.

However, in programming via constructive proofs, the specifications of the program are first proven to be satisfiable and the code is then extracted from the proof. As part of the MIZAR-C group at the University of Alberta, I have been investigating the feasibility of this programming methodology by using and extending the MIZAR-C system [23], a natural deduction proof environment that extracts Lisp code from constructive proofs.

In programming via constructive proofs, the relationship between specifications and programs is usually established by a specification language. In the case of MIZAR-C, the specification language is the language of limited second order predicate logic formulas. For example, we can relate input x to output y by reading the formula

$$\forall x\ \exists y\ st\ Post[x, y]$$

as a specification of a program which, when given an x, finds a y for which property $Post[x, y]$ on x and y holds.

Besides our MIZAR-C group, there are other research groups investigating the use of using constructive proofs to realize correct programs. NUPRL [3], COQ [5], Martin Löf's Type Theory [16], and the Calculus of Construction [4] use type systems as their formal language. They try to approach the problem of program derivation from the type perspective. On the other hand, PX [7, 8] is a system that uses a logic such that the programs exists as terms in the object language. A Lisp program can therefore be extracted from a constructive proof of a formula. While PX implements a particular logic, Elf [17] is a logical framework in which different logics can be implemented. MIZAR-C is a system which falls somewhere between PX and Elf as it can implement many different logics through the extension of inference rules and axioms. Like both, programs are generated from proofs through a realizability interpretation.

One of the distinctive features of MIZAR-C is that it attempts to account for the resources required to compute, whereas systems like NUPRL or COQ do not seem to address this problem. The fact that the organization of the proof affects the efficiency of the extracted program is well-known. For example, Manna and Waldinger [13] investigate program synthesis using deductive-tableau proofs and they emphasize the importance of the complexity of the synthesized programs. In MIZAR-C, statements proved by non-constructive reasoning are marked as non-computable and do not contain any computational content. Other statements are marked according to the resources consumed when their computational content is extracted. In this way, we can employ classical reasoning on the statements where computations are not required.

Other current researchers in this area include Brent Knight [12]. In his thesis, he points out that the programs extracted from constructive proofs are usually littered with redundant components. He then gives an algorithm for automatically pruning these useless subprograms.

1.3 The MIZAR-C System

The MIZAR-C system is a natural deduction proof environment that extracts code from constructive proofs. The language of the logic is closely related to that of the MIZAR-MSE language [9], a subset of MIZAR[19]. MIZAR-C's proof checking environment is implemented using the Synthesizer Generator [18], which provides an interactive syntax-directed editing environment for writing proof texts. The language for the realizations that make up the extracted programs is Lisp.

It was Kleene [1, 11] who proposed the idea of realizability which relates formulas to programs. Curry and Howard [4] showed an isomorphism between natural deduction logics and typed lambda calculi. In other words, they related propositions to types. The realizability interpretation of MIZAR-C, called *sparse realizability*, is based upon the Curry-Howard isomorphism, although the isomorphism is destroyed since it is no longer possible to convert programs back into their corresponding proofs.

Every logical formula has its corresponding computation under a full realizability interpretation. Such a computation is called the *realization* of that formula. However, not every realization has actual computational content. For example, some realizations have no inputs and so they are simply constants or expressions that require evaluation in order to produce the constant. Under sparse realizability, realizations are generated only for those formulas that have computational content. In other words, this sparse realization of formulas results in generated code that is optimized, since it only produces expressions that contribute to the overall computation.

1.4 Overview of Thesis

My main project was to implement binary tree data structure of bit strings in the MIZAR-C system. This can be seen as a continuation of the work done by Kippen [10]. In Kippen's thesis, she defines finite sequences of bit strings in MIZAR-C. I continued this research direction by implementing binary trees in the system. Our motivation was to investigate if it is feasible to represent binary trees in bits strings since the binary tree is a basic and useful data structure in programming. We also wanted to find out any existing errors in MIZAR-C as well as any incompleteness of the system.

This thesis contains six chapters in addition to the introduction. Chapter 2 contains some background material on the *Bits* machine, an extension for which the functions extracted from the proofs is applied to. Chapter 3 describes how binary trees are represented and defined in MIZAR-C. Chapter 4 discusses the proofs of some theorems for binary trees. Then, it discusses the definition of some functions for binary trees and motivates us to implement a new inference rule, recursive function definition. Chapter 5 discusses the new inference rule in detail and illustrates the use of the rule by a small example. Appendix A discusses the realization of the rule and compares it with that of the induction rule. Chapter 6 decribes the definitions of two recursive functions for binary trees in MIZAR-C and discusses their complexities. Appendices B and C shows their annotated proofs. Finally, Chapter 7 contains some future research directions in binary trees, as well as in the MIZAR-C system in general.

The reader may find the following information useful in reading this thesis. Every object in MIZAR-C is expressed in the form of:

```
<module_name>.<object_name>
```

This convention allows us to distinguish different objects having the same names defined in different modules. In this thesis, only the object names are mentioned except in the examples of statements and proofs done in MIZAR-C. For these examples, the following type style is used:

```
/* This is a comment line */
for x being Tx holds P[x];
```

Chapter 2

Background

2.1 Bits

The realizability interpretation of MIZAR-C alone has no computational power. Without any other extensions, the functions extracted from the proofs have no objects to apply to. The system has thus been extended with a basic data type called *Bits*, that is, bit strings of *0*'s and *1*'s. The fundamental building blocks are the bit strings *0*, *1*, and *nil*.

```
/* Definition of the type Bits */
given Bits being [Any ];

/* Define 0 & 1 to be the bit strings of length 1; */
/* and nil to be the empty bit string */
given 0, 1, nil being Bits;
```

A decider function is provided that determines if a given bit string is *nil* or not.

```
BA_nil_or_not: (for x being Bits holds
   ( (x = nil) or (x <> nil) );
```

Two length predicates are provided to compare the length of two different bit strings. *bits_len_lt[x,y]* means bit string x is shorter in length than bit string y; and *bits_len_eq[x,y]* means bit string x has the same length as bit string y.

```
given bits_len_lt being [Bits, Bits ];

given bits_len_eq being [Bits, Bits ];
```

A function that decides if a given bit (i.e. a bit string of length equal to 1) is a *0* or a *1* is provided.

BA_len_eq_1: (for x being Bits holds
(bits_len_eq[1, x] iff ((x = 1) or (x = 0))));

A function that, given two bit strings x and y such that y has length no less than x, determines if x is longer in length than y or x has the same length as y is provided.

BA_len_not_lt: (for x, y being Bits holds ((not bits_len_lt[x, y]) implies
(bits_len_lt[y, x] or bits_len_eq[x, y])));

The axioms stating the asymmetric and the transitive properties of *bits_len_lt* are provided.

BA_len_lt_asym: (for x, y being Bits holds
(bits_len_lt[x, y] implies (not bits_len_lt[y, x])));

BA_len_lt_trans: (for x, y, z being Bits holds (bits_len_lt[x, y] implies
(bits_len_lt[y, z] implies bits_len_lt[x, z])));

The axioms stating the reflexive, the symmetric and the transitive properties of *bits_len_eq* are also provided.

BA_len_eq_refl: (for x being Bits holds bits_len_eq[x, x]);

BA_len_eq_sym: (for x, y being Bits holds
(bits_len_eq[x, y] implies bits_len_eq[y, x]));

BA_len_eq_trans: (for x, y, z being Bits holds (bits_len_eq[x, y] implies
(bits_len_eq[y, z] implies bits_len_eq[x, z])));

Two primitive operations on *Bits* are provided to build new bit strings and to extract sub-strings from existing bit strings. The *cat* function concatenates two bit strings together, while the *split* function divides a bit string into two pieces modulo another bit string. Therefore, only the length of the second bit string is relevant to the *split* function. For example:

(cat < 1, 00 >) = 100
(split < 111, 0 >) = < 1, 11 >
(split < 111, 011 >) = < 111, nil >
(split < 111, 0101 >) = < 111, nil >

given cat being (<Bits, Bits > -> Bits);

BA_cat: (for x, y being Bits holds
(ex z being Bits st (z = (cat <x, y >))));

given split being (<Bits, Bits > -> <Bits, Bits >);

BA_split: (for x, y being Bits holds
(ex z1, z2 being Bits st ((split <x, y >) = <z1, z2 >)));

Two axioms on the relationships between *cat* and *split* are provided.

BA_cat_split: (for x, y being Bits holds
(x = (cat (split <x, y >))));

BA_split_cat: (for x, y being Bits holds
((split <(cat <x, y >), x >) = <x, y >));

Some other axioms (listed below) have also been added to the MIZAR-C system. An implementation has been written for those primitives that have associated computation.

BA_nil_not_1: (nil <> 1);

BA_nil_not_0: (nil <> 0);

BA_1_not_0: (1 <> 0);

BA_nil_or_not: (for x being Bits holds ((x = nil) or (x <> nil)));

BA_len_lt_nil: (for x being Bits holds
((x <> nil) iff bits_len_lt[nil, x]));

BA_len_eq_no_lt: (for x, y being Bits holds (bits_len_eq[x, y] iff
(not (bits_len_lt[x, y] or bits_len_lt[y, x]))));

BA_len_eq_1: (for x being Bits holds
(bits_len_eq[1, x] iff ((x = 1) or (x = 0))));

BA_cat_nil: (for x being Bits holds
(((cat <nil, x >) = x) & ((cat <x, nil >) = x)));

BA_cat_len_eq: (for x, y, z being Bits holds (bits_len_eq[y, z] iff
bits_len_eq[(cat <x, y >), (cat <x, z >)]));

BA_cat_len_lt: (for x, y, z being Bits holds (bits_len_lt[y, z] iff

```
    bits_len_lt[(cat <x, y >), (cat <x, z >) ]));

BA_cat_both_len_eq: (for x, y being Bits holds
    bits_len_eq[(cat <x, y >), (cat <y, x >) ]);

BA_cat_assoc: (for x, y, z being Bits holds
    ((cat <(cat <x, y >), z >) = (cat <x, (cat <y, z >) >)));

BA_split_1: (for x being Bits holds ( ((split <1, x >) = <1, nil >) or
    ((split <1, x >) = <nil, 1 >) ));

BA_split_eq: (for x, y, z being Bits holds (bits_len_eq[y, z ] implies
    ((split <x, y >) = (split <x, z >))));

BA_split_eq_rev: (for x, y, z being Bits holds
    ((not bits_len_lt[x, y ]) implies ((not bits_len_lt[x, z ]) implies
    (((split <x, y >) = (split <x, z >)) iff
    bits_len_eq[y, z ]))));

BA_split_big: (for x, y being Bits holds
    ((not bits_len_lt[y, x ]) iff ((split <x, y >) = <x, nil >)));

given LT being [Bits, Bits ];

BA_LT: (for x, y being Bits holds
    (bits_len_lt[x, y ] iff LT[x, y ]));
```

The idea was to add the minimal set of primitives such that all other desired functionality could be derived from them. In addition, a library of helper functions and theorems has been built to reduce the effort spent on re-proving the same function or theorem all the time.

We use *Bits* as the basic data type, so we need an access function to index into bits, to read and write individual bits. But to do that, we need an index type. So we use the length of a bit string as the index, which leads to *unary naturals*.

2.2 Unary Naturals

Unary naturals, or *unat*, is defined as the set of bit strings where every bit is a *0*, that is,

$$\{nil, 0, 00, 000, \ldots\}$$

The fundamental building blocks in *unat* are *U0* and *U1* with *U0* = *nil* and *U1* = *0*. *U0* is the *unat* representation of the natural number 0. It is not the bit string containing the bit *0*, so we call it *U0*. Similarly, *U1* is the *unat* representation of the natural number 1.

```
/* Define the type of U0 */
given U.U0 being Bits;

/* Implementation of U0 in Bits */
U.U0: ( U.U0 = nil);

/* Define the type of U1 */
given U.U1 being Bits;

/* Implementation of U1 in Bits */
U.U1: (U.U1 = 0);
```

In order to define *unat* formally in MIZAR-C, we need to introduce the function *index*. We derived the function *index* which takes in two bit strings *x* and *i* and returns the $(i+1)^{th}$ bit of *x* if there is one. Otherwise, it returns *nil*. For example:

((index 101) 0) = 0
((index 0) 101) = nil

```
/* State the domain and range of index */
given U.index being (Bits -> (Bits -> Bits));

/* Specification of index */
U.index: (for x being Bits holds
   (for i being Bits holds
      (ex x1, x2, x3 being Bits st
         ( ((split <x, i >) = <x1, x2 >) & ((split <x2, U.U1 >) = <((U.index x) i), x3 >)))));
```

The formal definition of *unat* is shown below:

```
U.unat_def: define U.unat of x being Bits by
   (for y being Bits holds
      (bits_len_lt[y, x ] implies (((U.index x) y) = 0)));
```

Note that recursive definition of unat[x] is not required, since by using the universal quantifier in the direct definition, we can iterate the test for a *0* bit over each bit position.

We wanted to have a function that gives the length of a bit string. Since we do not have a built-in method for counting, we expressed the length of a bit string in *unat* using the *length* function. In other words, *length* converts every single bit of a bit string to a *0*. For example:

(length 101) = 000

```
given U.length being (Bits -> Bits);

U.length: (for x being Bits holds
   ( bits_len_eq[x, (U.length x)] & U.unat[(U.length x)] ));
```

The way that we derived the *length* function can be seen as definition by properties. Instead of actually implementing the function (as we usually do in traditional programming), we proved its properties. This is one of the distinctive features of programming via constructive proofs.

2.3 Finite Sequences

Another more complex structure in MIZAR-C is *finite sequences* of *Bits*. Before explaining how this structure is represented, we have to introduce some relevant functions and definitions.

The function *nozab* returns the number of zeros at beginning of a bit string. For example:

(nozab 00101) = 00
(nozab 101) = nil

```
given fseq.nozab being (Bits -> Bits);

fseq.nozab: (for x being Bits holds
   (ex z1, z2 being Bits st
      ( U.unat[(fseq.nozab x)] &
      ((split <x, (fseq.nozab x)>)=<(fseq.nozab x), z1>) &
      ( (z1=nil) or ((split <z1, U.U1>)=<1, z2>) ) )));
```

A *Packet* is defined as a structure consisting of a bit string preceded by its length. A *1* bit is used as a delimiter between the length and the bit string. For example:

> *0001xxx*
> is the *Packet* of the bit string xxx, which has length 000.
> *1*
> is the *Packet* of the *nil* bit string, which has length *nil*.

The following is the formal definition of *Packet* used in MIZAR-C:

```
packet_def: define Packet of x being Bits by
   (ex z1, z2 being Bits st
      ( ((split <x, (fseq.nozab x) >) = <(fseq.nozab x), z1 >) &
      ((split <z1, U.U1 >) = <1, z2 >) &
      bits_len_eq[(fseq.nozab x), z2 ] ));
```

The function *first* returns the first *Packet* of a bit string, if there is one. Otherwise, the function returns the original bit string. For example:

$$(first\ 0001101010) = 0001101$$
$$(first\ 000110) = 000110$$

```
given fseq.first being (Bits -> Bits);

fseq.first: (for x being Bits holds
   (ex z1, z2, z3, z4, z5 being Bits st
      ( ((split <x, (fseq.nozab x)>)=<(fseq.nozab x), z1>) &
      ((split <z1, U.U1>)=<z2, z3>) &
      ((split <z3, (fseq.nozab x)>)=<z4, z5>) &
      ((fseq.first x)=(cat <(cat < (fseq.nozab x), z2>), z4>)) )));
```

The function *rest* performs exactly what its name implies. *(rest x)* returns the bit string that remains once *(first x)* has been removed from x. For example:

$$(rest\ 0001101010) = 010$$
$$(rest\ 000110) = nil$$

```
given fseq.rest being (Bits -> Bits);

fseq.rest: (for x being Bits holds
   (ex z1, z2, z3, z4 being Bits st
      ( ((split <x, (fseq.nozab x)>)=<(fseq.nozab x), z1>) &
      ((split <z1, U.U1>)=<z2, z3>) &
      ((split <z3, (fseq.nozab x)>)=<z4, (fseq.rest x)>) )));
```

$$(((Iter\ f)\ (length\ n))\ x) = \underbrace{(f\ (f \cdots (f}_{(length\ n)\ \text{times}}\ \ x)) \cdots)$$

Figure 2.1: The *Iter* function

The function *Iter* takes in

- *f*, a total function on *Bits*
- *n*, the number of times that *f* is to be composed
- x, a bit string

and returns the result after applying *f* to x *n* times (Figure 2.1). To simplify the definition of the *Iter* function, we express *n* in *unat* using the *length* function.

```
given fseq.Iter being ((Bits -> Bits) -> (Bits -> (Bits -> Bits)));

fseq.Iter: (for f being (Bits -> Bits) holds
   ((for x being Bits holds ((f x)=(f x))) implies
   (for n being Bits holds
      (for x being Bits holds
         ( ((((fseq.Iter f) (U.length nil)) x)=x) &
         ((((fseq.Iter f) (U.length (cat <n. 0 >))) x)
         =(f (((fseq.Iter f) (U.length n)) x))) ))))));
```

The *Iter* function was initially derived to define recursive structures by implementation. This definition by implementation approach introduced in Kippen's thesis [10] defines recursive structures directly and gives an iterative test for correctness of construction.

$$\underbrace{0\ 1\ 0}_{i}\ \overbrace{0\ 1\ 1}^{ii}\ \underbrace{0\ 0\ 1\ 1\ 0}_{iii}$$

i. first Packet containing the bit string 0
ii. second Packet containing the bit string 1
iii. third Packet containing the bit string 10

Figure 2.2: Analysis of the bit string representing the finite sequence {0,1,10}

Let us go back to the definition of a finite sequence in MIZAR-C. Since we have to express a finite sequence as a bit string, we need to be able to recover the individual bit strings that make up the sequence. Therefore, each bit string is wrapped in a *Packet*, and thus a finite sequence is defined as a sequence of *Packets*. For example:

01001100110
represents the finite sequence {*0,1,10*}

Figure 2.2 shows a detailed analysis of the bit string representing the finite sequence.
Finally, the formal definition of a finite sequence in MIZAR-C is shown below:

```
finseq_def: define FinSeq of x being Bits by
   (for n being Bits holds
      ( ((fseq.first (((fseq.Iter fseq.rest) (U.length n)) x)) = nil) or
      fseq.Packet[(fseq.first (((fseq.Iter fseq.rest) (U.length n)) x)) ] ))
```

Chapter 3

Binary Trees in Mizar-C

3.1 Representation of a Binary Tree

In programming, a binary tree is usually represented by a 3-tuple consisting of a data item stored in the node, and pointers to the left and the right sub-trees. Each sub-tree is either a binary tree or empty. For example, in Pascal, the declaration of the type BTree is as follows:

```
type BTree = ^TreeRep;

TreeRep = record
                 Root : ElemType;
                 Left : BTree;
                 Right: BTree
           end; { TreeRep }
```

Since we are working with *Bits*, we do not have the notion of pointer. Direct implementation of the data structure seems to be impossible. However, this gives us some intuition on how a binary tree should be represented. Let us consider the following example.

Node	Data
n_1	*0*
n_2	*1*
n_3	*10*

Table 3.1: Data stored in nodes n_1 to n_3

Table 3.1 shows the data stored in each of the nodes n_1 to n_3. In general, the data can be any bit string of finite length.

Figure 3.1 shows a binary tree T_1 consisting of a single node n_1 only. We can view T_1 to be a binary tree consisting of a root node n_1 with data *0*, an empty left sub-tree

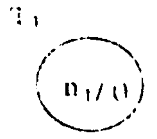

Figure 3.1: Binary tree T_1

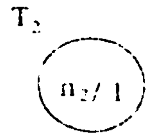

Figure 3.2: Binary tree T_2

and an empty right sub-tree. Similarly, Figure 3.2 shows a binary tree T_2 consisting of a single node n_2. Figure 3.3 then shows a binary tree T_3 consisting of a root node n_3, a left sub-tree T_1, and a right sub-tree T_2.

Intuitively, the pointer-free representation of T_3 is:

<10, <0, nil, nil>, <1, nil, nil>>

where

- *10* represents the data stored in n_3,
- *<0, nil, nil>* represents T_1, and
- *<1, nil, nil>* represents T_2.

However, our existing *Bits* extension only allows types *Bits* and tuples of *Bits*. The representation of T_3 above is far more complicated than just tuples of *Bits*.

In the previous chapter, we introduced a data structure called finite sequences. This structure gives us a way to store any finite number of items in a bit string using *Packets*. So, a finite sequence with 3 *Packets* is a way to represent a binary tree, with the first *Packet* representing the wrapped data item, the second the wrapped left sub-tree and the third the wrapped right sub-tree. In our previous example, the bit string representing T_1 is shown in Figure 3.4. The data *0* is wrapped in the first *Packet* while the empty left and the empty right sub-trees are wrapped in the second and the third *Packets* respectively.

Similarly, the bit string representing T_2 is shown in Figure 3.5. The data *1* is wrapped in the first *Packet* while the empty left and the empty right sub-trees are wrapped in the second and the third *Packets* respectively.

Knowing the representation of T_1 and T_2, the bit string representing T_3 is shown in Figure 3.6. The data *10* is wrapped in the first *Packet* while the left sub-tree T_1 and the right sub-tree T_2 are wrapped in the second and the third *Packets* respectively.

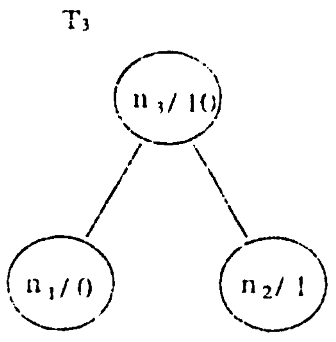

Figure 3.3: Binary tree T_3

$$\underbrace{0\ \ 1\ \ 0}_{i}\ \ \overbrace{1}^{ii}\ \ \underbrace{1}_{iii}$$

i. wrapped data *0*
ii. wrapped empty left sub-tree
iii. wrapped empty right sub-tree

Figure 3.4: Bit String representing T_1

3.2 Some Useful Functions

For ease of dealing with binary trees, we derived the following functions in order to create a *Packet*, to retrieve the data from a *Packet*, and to retrieve the wrapped data, the left and the right sub-trees from a binary tree.

The function *pack* wraps a bit string in a *Packet*. Technically, it concatenates a bunch of zeros with length equal to the bit string, a *1* as a delimiter, and the bit string together. For example:

(pack 10) = 00110
(pack nil) = 1

```
given bt.pack being (Bits -> Bits);

bt.pack: (for x being Bits holds
  ((bt.pack x)=(cat <(U.length x), (cat <1, x>)>)));
```

How did we get the above? In the proof text below, *pack_spec* is the specification of the *pack* function. The proof of the *pack_spec* results in its implementation. We then extract and name the function in *pack_extract* and *pack_name* respectively.

$$\underbrace{0\ 1\ 1}_{\text{i}}\ \overbrace{1}^{\text{ii}}\ \underbrace{1}_{\text{iii}}$$

i. wrapped data *1*
ii. wrapped empty left sub-tree
iii. wrapped empty right sub-tree

Figure 3.5: Bit String representing T_2

$$\underbrace{0\ 0\ 1\ 1\ 0}_{\text{i}}\ \overbrace{0\ 0\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 1\ 1}^{\text{ii}}\ \underbrace{0\ 0\ 0\ 0\ 0\ 1\ 0\ 1\ 1\ 1\ 1}_{\text{iii}}$$

i. wrapped data *10*
ii. wrapped left sub-tree T_1
iii. wrapped right sub-tree T_2

Figure 3.6: Bit String representing T_3

```
now
   /* Proof of pack_spec */
   let x be Bits;

   consider z being Bits such that
   1: (z = (cat <1, x >)) by elim[1,x](BA_cat);
   2: ( bits_len_eq[x, (U.length x) ] & U.unat[(U.length x) ] ) by elim[x](U.length);
   consider z' being Bits such that
   (z' = (cat <(U.length x), (cat <1, x >) >))
      by elim[(U.length x),(cat <1, x >) ](BA_cat, _PREVIOUS, 1);
   thus (ex y being Bits st
      (y = (cat <(U.length x), (cat <1, x >) >))
   ) by exintro(_PREVIOUS);

end;


/* Specification of pack */
pack_spec: (for x being Bits holds
   (ex y being Bits st
      (y = (cat <(U.length x), (cat <1, x >) >))
   )
) by direct(_PREVIOUS);
```

```
/* Extract the pack function */
pack_extract: (ex pack being (Bits -> Bits) st
   (for x being Bits holds
      ((pack x) = (cat <(U.length x), (cat <1, x >) >))
   )
) by choice(_PREVIOUS);

/* Name the pack function */
consider pack being (Bits -> Bits) such that pack_name: (for x being Bits holds
   ((pack x) = (cat <(U.length x), (cat <1, x >) >)) ) by direct(_PREVIOUS);
```

As opposed to the way that we derived the *length* function, the way that we derived the *pack* function can be seen as definition by implementation. Instead of proving the properties of *pack*, we specify the function directly.

On the other hand, the function *unpack* performs exactly the opposite to what the *pack* function does. (*unpack x*) returns the unwrapped bit string if *x* is a *Packet*. Technically, it removes all the leading zeros and then removes the leading *1* if there is any. For example:

(*unpack 00110*) = *10*
(*unpack 1*) = *nil*
(*unpack 0011*) = *1*

```
given bt.unpack being (Bits -> Bits);

bt.unpack: (for x being Bits holds
   (ex z1', z2' being Bits st
      ( ((split <x, (fseq.nozab x)>)=<(fseq.nozab x), z1'>) &
      ((split <z1', U.('1>)=<z2', (bt.unpack x)>) )));
```

The function *data* does exactly what the *first* function does. We define this function again because we want to have a more meaningful function when working with binary trees. If the input bit string is a binary tree, the *data* function returns the wrapped data of the root. For example:

(*data 01011*) = *010*
(*data 01100*) = *011*
(*data 00011*) = *00011*

```
given bt.data being (Bits -> Bits);

bt.data: (for x being Bits holds
   ((bt.data x)=(fseq.first x)));
```

The function *Ltree* takes in a bit string *x* and returns the left sub-tree of *x* if *x* is a binary tree. For example, consider the binary tree T_3 in the previous section, we have:

$$(l\_tree\ T_3) = T_1$$

```
given bt.l_tree being (Bits -> Bits);

bt.l_tree: (for x being Bits holds
   ((bt.l_tree x)=(bt.unpack (fseq.first (fseq.rest x)))));
```

Similiar to the function *l_tree*, the function r_tree takes in a bit string x and returns the right sub-tree of x if x is a binary tree. For example, consider the binary tree T_3 again, we have:

$$(r\_tree\ T_3) = T_2$$

```
given bt.r_tree being (Bits -> Bits);

bt.r_tree: (for x being Bits holds
   ((bt.r_tree x)=(bt.unpack (fseq.rest (fseq.rest x)))));
```

3.3 The "walk" Function

After deciding how to represent a binary tree in MIZAR-C and defining some relevant functions, we have to find a way to define a binary tree formally in the system. The basic intuition is clearly to define a binary tree recursively as follows:

```
bt.btree_def: define bt.btree of x being Bits by
   ( fseq.Packet[(bt.data x)] &
   ( fseq.Packet[(bt.l_tree x)] & ((bt.l_tree x)=nil) or bt.btree[(bt.l_tree x)] ) &
   ( fseq.Packet[(bt.r_tree x)] & ((bt.r_tree x)=nil) or bt.btree[(bt.r_tree x)] ) );
```

That is, a binary tree is composed of a node, a left sub-tree and a right sub-tree, while each sub-tree is either a binary tree or empty.

However, this version of MIZAR-C does not have the ability to define recursive structures directly. The reason is to avoid the possibility of the introduction of contradictions by the definition. For example, from the definition below:

```
define P of x being Any by (not P[x]);
```

we can derive contradictions, and thus introduce whatever we want into the system. To avoid such contradictions, syntactic rules need to be used — the one in MIZAR-C is that a definition cannot mention the thing being defined, that is, definitions are macros.

Another possible way to define a binary tree is to describe how to test if a bit string is a correctly formed binary tree. That is, to give an iterative test for the correctness of construction using the *Iter* function. The only way of testing a data structure is to go through it sequentially. For a list, this is easy. We can simply look at the head and then traverse the tail. But for a tree, this means that we have to look at every possible path. How do we do this walk?

We want to derive the function *walk* to take in

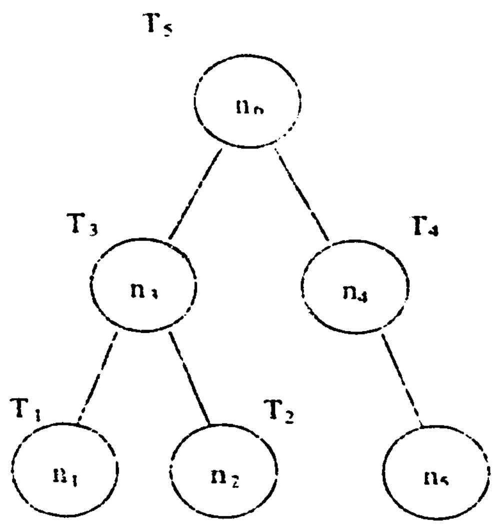

Figure 3.8: Binary tree T_5

- *(walk <T_5,01>) = <T_3,1>*
  The function returns the left sub-tree T_3 and the remaining path *1*.
- *(walk <T_3,1>) = <T_2,nil>*
  The function returns the right sub-tree T_2 and the empty path *nil*.
- *(walk <T_2,nil>) = <T_2,nil>*
  The function returns the original tree T_2 and the original empty path *nil*.
- *(walk <T_5,101>) = <T_4,01>*
  The function returns the right sub-tree T_4 and the remaining path *01*.
- *(walk <T_4,01>) = <T_4,01>*
  The function returns the original tree T_4 and the original path *01*.

3.4 Eliminating Tuples from the "walk" Function

The *walk* function discussed above is of type:

<Bits, Bits> -> <Bits, Bits>

that is, from a tuple to a tuple. We use tuples to handle input and output parameters of a function such as *walk*. In general, tuples are the method of choice for multiple input and output parameters.

An alternative to pass in multiple parameters instead of using tuples is to apply the *choice* rule multiple times. The *choice* rule allows the introduction of a function

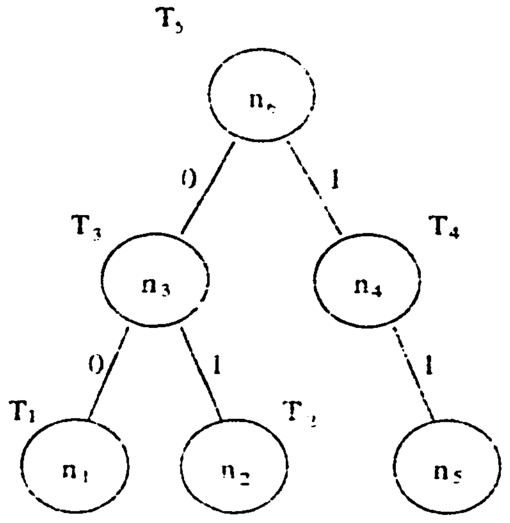

Figure 3.9: Binary tree T_5 with labels on the branches

from a proof of its specifications. Its proper form should take a statement of existence, plus one of uniqueness:

$$\frac{\begin{array}{l}\textit{for x being TYPE1 holds (ex y being TYPE2 st P[x,y]),}\\ \textit{for x being TYPE1 holds (for y, z being TYPE2 holds P[x,y] \& P[x,z] implies y = z)}\end{array}}{\textit{ex f being TYPE1 -> TYPE2 st (for x being TYPE1 holds P[x, (f x)])}}$$

For example, if we want to derive a function that takes in two bit strings and returns the length of the longer one (if their length are the same, just simply return the length of any one of them), we can use the following method:

```
now
  let x be Bits;

  now
    let y be Bits;
    ⋮
    thus (ex z being Bits st
      ( (z = z) &
      (bits_len_lt[x, y ] implies (z = (U.length y))) &
      ((not_bits_len_lt[x, y ]) implies (z = (U.length x))) )
    );

  end;
```

```
    thus (ex f being (Bits -> Bits) st
      (for y being Bits holds
        ( ((f y) = (f y)) &
        (bits_len_lt[x, y ] implies ((f y) = (U.length y))) &
        ((not bits_len_lt[x, y ]) implies ((f y) = (U.length x))) )
      )
    ) by choice(_PREVIOUS);

  end;


  (ex max being (Bits -> (Bits -> Bits)) st
    (for x being Bits holds
      (for y being Bits holds
        ( (((max x) y) = ((max x) y)) &
        (bits_len_lt[x, y ] implies (((max x) y) = (U.length y))) &
        ((not bits_len_lt[x, y ]) implies (((max x) y) = (U.length x))) )
      )
    )
  ) by choice(_PREVIOUS);
```

However, we have no other direct way to pass out multiple parameters without using tuples. The *choice* rule simply only introduces functions that can pass out one output parameter.

Although the use of tuples seems to be a good way to derive the *walk* function, we finally decided to use another approach because of the following two reasons. First, the *Iter* function is instantiated only for functions whose domains are *Bits*. Instantiating another iterator function that works for functions on <*Bits*,*Bits*> domain requires re-proving many theorems. This is easy in principle but extremely tedious. Second, we want to see if it is feasible to use *Bits* to handle functions with multiple input and output parameters.

$$\underbrace{0\ 0\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 1\ 1}_{\text{i}}\ \overbrace{0\ 0\ 1\ 0\ 1}^{\text{ii}}$$

i. wrapped binary tree *01011*
ii. wrapped path *01*

Figure 3.10: Analysis of the finite sequence consisting of the binary tree *01011* and the path *01*

Since the *walk* function takes in and returns two parameters, a binary tree and a path, we wrap each of them up using a *Packet* and then concatenate them together to form a finite sequence with two *Packets*. We call this finite sequence a ***tree-path finite sequence***. For example, if we want to apply the *walk* function to the tree *01011* and the path *01*, we pass in the following bit string to the *walk* function:

0000010101100101

Figure 3.10 shows a detailed analysis of the bit string.

On the other hand, we also need some way to recover the binary tree and the path from a tree-path finite sequence. The following functions were derived to simplify this process.

The function *tree* takes in a bit string x and returns the binary tree in x if x is a tree-path finite sequence. Technically, the function unwraps the first *Packet* if there is one. For example:

(tree 0000010101100101) = 01011

```
given bt.tree being (Bits -> Bits);

bt.tree: (for x being Bits holds
   ((bt.tree x)=(bt.unpack (fseq.first x))));
```

Similar to the *tree* function, the function *path* takes in a bit string x and returns the path in x is x is a tree-path finite sequence. Technically, the function unwraps the second *Packet* if there is one. For example:

(path 0000010101100101) = 01

```
given bt.path being (Bits -> Bits);

bt.path: (for x being Bits holds
   ((bt.path x)=(bt.unpack (fseq.rest x))));
```

Finally, we derived the function *walk* formally in MIZAR-C as follows:

```
given bt.walk being (Bits -> Bits);

bt.walk: (for x being Bits holds
   ( ((bt.walk x)=(bt.walk x)) &
   (ex step1, o_path1 being Bits st
      ( ((split <(bt.path x), U.U1>)=<step1, o_path1>) &
      ((step1=nil) implies ((bt.walk x)=x)) &
      ((step1=0) implies
         ( (((bt.l_tree (bt.tree x))=nil) implies ((bt.walk x)=x)) &
         (((bt.l_tree (bt.tree x))<>nil) implies
            ((bt.walk x)=(cat <(bt.pack (bt.l_tree (bt.tree x))), (bt.pack o_path1)>))) )) &
      ((step1=1) implies
         ( (((bt.r_tree (bt.tree x))=nil) implies ((bt.walk x)=x)) &
         (((bt.r_tree (bt.tree x))<>nil ) implies
            ((bt.walk x)=(cat <(bt.pack (bt.r_tree (bt.tree x))), (bt.pack o_path1 )>)))))))));
```

3.5 Definition of Binary Trees

Once the *walk* function was derived, the definition of binary trees in MIZAR-C could be stated accordingly. We define a binary tree as a bit string such that no matter which path we walk down the tree, the resulting bit string is still a finite sequence with three *Packets*. The definition is a bit different from our usual understanding of a binary tree. In order to verify that our definition corresponds to our intuitive recursive understanding of a binary tree, we need to prove that our definition is isomorphic to the usual accepted one. However, the proof is outside the scope of this thesis.

The following is the formal definition of a binary tree in MIZAR-C:

```
bt.3_packets_def: define bt.3packets of x being Bits by
  ( fseq.Packet[(fseq.first x)] &
  fseq.Packet[(fseq.first (fseq.rest x))] &
  fseq.Packet[(fseq.rest (fseq.rest x))] );

bt.btree_def: define bt.btree of x being Bits by
    (for n being Bits holds
       bt.3packets[(bt.tree (((fseq.Iter bt.walk) (U.length n))
         (cat <(bt.pack x), (bt.pack n)>)))]);
```

Suppose a bit string is a binary tree, applying the *walk* function to it with any path will return the original binary tree, its left sub-tree or its right sub-tree. Since each of the left and the right sub-trees of a binary tree is again a binary tree, applying the *walk* function to the bit string any number of times with any path will still yield a finite sequence with three *Packets*. On the other hand, if a bit string satisfies the definition, we can actually re-construct the binary tree by breaking down the finite sequence into the root, the left sub-tree and the right sub-tree. We can continue this process until the whole tree is being re-constructed because each of the left and the right sub-trees is a finite sequence with three *Packets* by the definition. Hence, the binary tree definition captures our intent.

```
U.unat_def: define U.unat of x being Bits by
  (for y being Bits holds
     (bits_len_lt[y, x ] implies (((U.index x) y) = 0)));
```

Figure 3.11: Definition of Unary Naturals

It is interesting to compare the definition of binary trees to that of unary naturals and that of finite sequences. Although *unats* are recursive objects, it is not necessary to use recursion in order to define them. In the definition (Figure 3.11), the universally quantified variable acts as an index into to the *unat* structure. Therefore, the universal quantifier provides the mechanism for iterating over the structure.

```
finseq.def: define FinSeq of x being Bits by
  (for n being Bits holds
    ( ((fseq.first (((fseq.Iter fseq.rest) (U.length n)) x)) = nil) or
    fseq.Packet[(fseq.first (((fseq.Iter fseq.rest) (U.length n)) x)) ] ));
```

Figure 3.12: Definition of Finite Sequences

On the other hand, the definition of finite sequences (Figure 3.12) uses a *unat* as the index. The *Iter* function, together with the *rest* function, provides the mechanism for iterating over the structure.

Finally, the definition of binary trees uses a binary bit string as the path and its length as the index. The *Iter* function, together with the *walk* function, provides the mechanism for iterating over the structure.

Chapter 4

More on Binary Trees

4.1 Some Theorems on Binary Trees

Once binary trees were defined, the theory of binary trees needed to be proven. Several theorems are necessary to define the properties of binary trees. Two of the most important are the *recursive definition* of a binary tree and the *binary tree decider*. We had chosen to prove the recursive definition first and then the decider because of the following reasons:

- We do not want to refer to the original definition of a binary tree every time since the definition is too complicated.
- We can make use of the recursive definition to simplify our proof of the decider.

4.1.1 Recursive Properties

We want to prove a theorem re-stating the definition of a binary tree in its recursive form. This is important because it is the proof that establishes that our definition captures the recursive properties. Moreover, we state properties and algorithms on binary trees recursively all the time. Our initial attempt was to state the theorem using the *walk* function as follows:

```
(for x being Bits holds
   (bt.btree[x] iff (bt.3packets[x] &
   bt.btree[(bt.tree (bt.walk (cat <(bt.pack x), (bt.pack 0)>)))] &
   bt.btree[(bt.tree (bt.walk (cat <(bt.pack x), (bt.pack 1)>)))])));
```

The major advantage of stating the theorem in this way is that empty sub-trees and non-empty sub-trees do not need to be considered in different cases explicitly. However, its disadvantage dominates the advantage in this case. First, we do not want to refer back to the *walk* function every time since this function is too "low level" for binary trees. Second, this definition does not help in proving the decider that we are going to prove later.

We finally decided to state the recursive definition in the following way:

```
th: (for x being Bits holds
   (bt.btree[x] iff (bt.3packets[x] &
   (((bt.l_tree x) = nil) or bt.btree[(bt.l_tree x)]) &
   (((bt.r_tree x) = nil) or bt.btree[(bt.r_tree x)]))));
```

The attempt to prove this theorem outlined many sub-theorems that were useful. As binary trees are defined in terms of *Packets*, it was necessary to prove some thorems on operators related to *Packets* as follows:

```
b1.unpack_pack: (for x being Bits holds
   ((bt.unpack (bt.pack x)) = x));

b1.Packet_pack: (for x being Bits holds
   fseq.Packet[(bt.pack x)]);

b1.pack_unpack: (for x being Bits holds
   (fseq.Packet[x] implies ((bt.pack (bt.unpack x)) = x)));

b1.tree_extract: (for x, y being Bits holds
   ((bt.tree (cat <(bt.pack x), (bt.pack y)>)) = x));

b1.path_extract: (for x, y being Bits holds
   ((bt.path (cat <(bt.pack x), (bt.pack y)>)) = y));
```

The theorem *unpack_pack* states that if we wrap a bit string in a *Packet* and then unwrap it, the result will be the original bit string for all bit strings. The theorem *Packet_pack* states that every wrapped bit string is a *Packet*. The theorem *pack_unpack* states that if we unwrap a *Packet* and then wrap it, the result will still be the original *Packet*. The theorem *tree_extract* states that if we turn a binary tree and a path into a tree-path finite sequence and then extract the tree component of the sequence, the result will be the original tree. Lastly, the theorem *path_extract* states that if we turn a binary tree and a path into a tree-path finite sequence and then extract the path component of the sequence, the result will be the original path.

4.1.2 Decider

Given the recursive definition of a binary tree, we can prove the binary tree decider stated as follows:

```
b2.btree_decide: (for x being Bits holds
   ( bt.btree[x] or (not bt.btree[x]) ));
```

Proven constructively, this theorem provides a program that decides whether a given bit string is a proper binary tree.

The proof was by induction on the length of a bit string. However, in order to use the induction hypothesis, the following were proven:

b2.l_tree_lt_x: (for x being Bits holds
  ((x <> nil) implies bits_len_lt(bt.l_tree x), x]));

b2 r_tree_lt_x: (for x being Bits holds
  ((x <> nil) implies bits_len_lt[(bt.r_tree x), x]));

The above states that both the left sub-tree and the right sub-tree of a bit string are shorter than the bit string itself for any non-empty bit string.

4.2 Deriving Functions for Binary Trees

Once some basic theorems on binary trees were proven, our next step was to derive some functions for binary trees. One of the most basic functions that we want to derive is the *in-order traversal* function, or simply the *traversal* function, which gives the in-order traversal of all the nodes in a binary tree.

4.2.1 To Derive the Traversal Function

In a Pascal-like programming language, the traversal function is usually implemented using the recursive approach,

```
procedure traverse(t: link);
  begin
  if t<>nil then
    begin
    traverse(t^.l);
    printnode(t);
    traverse(t^.r)
    end
  end;
```

or the stack-based approach,

```
procedure traverse(t: link);
  begin
  exitflag := FALSE;
  push(t);
  repeat
    t := pop;
    if t=nil then
      if stackempty then
        exitflag := TRUE
      else
        printnode(pop)
    else
```

```
        begin
        push(t^.r);
        push(t);
        push(t^.l)
        end
    until exitflag
    end;
```

The recursive approach precisely mirrors the definition of traversal: "if the tree is nonempty, first traverse the left subtree, then print the root, then traverse the right subtree." This implementation seems very straightforward, but our question is whether we can implement the traversal function in this way using our existing inference rules. We want to prove some statement similar to the following in MIZAR-C:

```
ex trav being (Bits -> Bits) st
  (for x being Bits holds
    (((not bt.btree[x]) implies (z=nil)) &
    (bt.btree[x] implies
      (z=(cat <(cat <(trav (bt.l_tree x)), (bt.data x)>), (trav (bt.r_tree x))>)))));
```

The first conjunct in the specifications takes care of the situations where the bit string is either not a binary tree or an empty one. In this way, we can derive *trav* as a total function on the domain *Bits*. The second conjunct states the recursive definition of the function.

The most intuitive way to prove the above is to use the *choice* rule (Section 3.4) as it allows the introduction of a function from a proof of its specifications. However, there is no way to derive recursive functions by this rule. In the above case, the *choice* rule does not allow us to extract the *trav* function with some term *(trav (l_tree x))* (or *(trav (r_tree x))*) in the specifications.

The second method involves a stack. The stack is assumed to be initialized outside this procedure. This implementation works by saving the right subtree on a stack, then the root, then the left subtree. In this way, the in-order traversal of the tree will always be preserved inside the stack. In MIZAR-C, the stack-based approach can be implemented by applying the iterator function to the stack.

4.2.2 To Derive Recursive Functions

Although we could implement the traversal function in MIZAR-C using the stack-based approach, we finally decided to derive it using the recursive approach as this seems to be more natural. The recursive implementation of tree traversal is more natural than the stack-based implementation because *"trees are recursively defined structures and traversal is a recursively defined process"* [20].

Our first attempt was to derive the traversal function by using strong induction. We wanted to prove:

```
ex f being (Bits -> Bits) st
   (for x being Bits holds spec[f, x]);
```

where *spec[f, x]* is the specifications of the traversal function *f*. However, strong induction only lets us prove statements in the form of a leading universal quantifier:

```
for x being Bits holds P[x];
```

and so could not be used in deriving recursive functions in general.

Our second attempt was to derive a *fixed point operator* similar to the *partial recursive function* in [2]. A partial recursive function is written as $fix(f, x, F)$ where F is an expression with $fix(f, x, F)[a] = F(x := a, f := fix(f, x, F))$ for any expression a. For example, if we interpret $F(f, x)$ as

```
(cat <(cat <(f (l_tree x)), (data x)>), (f (r_tree x))>),
```

$fix(f, x, F)$ will give us the traversal function (recursive portion). However, we face the same problem in defining such an operator as in defining the traversal function recursively. This motivated us to invent a new inference rule recursive function definition.

Chapter 5

Recursive Function Definition

5.1 The New Inference Rule

A binary relation $\prec$ is a *partial order* if for all x, y, z,

$$x \not\prec x$$

$$x \prec y \land y \prec z \Rightarrow x \prec z$$

In other words, a binary relation is a partial order if it is irreflexive and transitive. We can interpret $x \prec y$ as "x is simpler than y". The first clause guarantees that we have no loops $x \prec x$, and in combination with the second assures that we have nothing of the form

$$x_0 \prec x_1 \prec \ldots \prec x_n \prec x_0$$

However, being a partial order is not sufficient to guarantee that recursion can be performed over the ordering. We also need the partial order to be *well founded.* A partial ordering $\prec$ over a set A is well founded if and only if there are no sequences $< x_n >_n$ in A so that

$$\ldots x_{n+1} \prec x_n \prec \ldots \prec x_1 \prec x_0$$

Such a sequence is called an *infinite descending chain.*

The well founded partial ordering guarantees that for every element in the type, all sequences induced by the ordering that start at the element lead to some *minimal element.* A minimal element is an element for which no element is "simpler than" it. In MIZAR-C, the order *bits_len_lt* is well founded over the built-in type *Bits*, with the element *nil* being the minimal element.

In MIZAR-C we want to be able to introduce recursive functions. However, if we allow the introduction of a recursive function, we have to be very careful to check that each subsequent call to the function is applied to elements that are simpler than the one before. It seems to be sufficient that this ordering on the elements be a well founded partial order. Thus we can allow the introduction of a recursive function as long as a proof of the well foundedness of the partial order is supplied.

We define the *recursive function definition* rule, or simply *recursive definition*, that allows the introduction of a recursive function from a proof of its specifications, as follows:

```
WellFounded[TYPE, LT],
for x being TYPE holds
  (for f being (TYPE -> TYPE) holds
    ((for y being TYPE holds (LT[y,x] implies Spec[f, y, (f y)])) implies
      (ex z being TYPE st Spec[f, x, z])))
-----------------------------------------------------------------------
ex f being (TYPE -> TYPE) st (for x being TYPE holds Spec[f, x, (f x)])
```

This rule is allowed on any type using any well founded partial order for that type.

We want to verify that the *recursive definition* rule is correct by analyzing the structure of the proof. In general, a *recursive definition* proof looks like the following:

```
now
  let x be Any;

  now
    let f be (Any -> Any);
    assume RecHyp: (for y being Any holds
      (LT[y, x ] implies Spec[f, y, (f y) ]))
    : } /* inside refer to objects simpler than x */
    thus (ex z being Any st
      Spec[f, x, z]);
  end;

  thus (for f being (Any -> Any) holds
    ((for y being Any holds
      (LT[y,x] implies Spec[f, y, (f y)]))
    implies (ex z being Any st
      Spec[f, x, z]))) by direct(_PREVIOUS);

end;

(ex f being (Any -> Any) st
  (for x being Any holds
    Spec[f, x, (f x)])) by recdef(_PREVIOUS);
```

where *Spec[f, x, z]* is the specification of the recursive function. The object *z* is usually defined in terms of the function *f* and the object *x*. When *z = (f x)*, *Spec[f, x, z]* gives the specifications of a recursive function. For example, we can define the specifications of the traversal function as follows:

```
define trav_spec of f being (Bits -> Bits), x being Bits, z being Bits by
  ((z=z) &
  ((not bt.btree[x]) implies (z=nil)) &
  (bt.btree[x] implies (z=(cat <(cat <(f (bt.l_tree x)), (bt.data x)>), (f (bt.r_tree x))>)) ));
```

Note that we have included *(z=z)* in the specifications so as to guarantee that the function is total on *Bits*. Otherwise, if the range of the function is not a subset of its domain, the function will be undefined for some input as computing it recursively is not possible. In general, this certificate of a total function is required for all specifications in using *recursive definition*.

We want to conclude that there exists a function f satisfying the specifications for all x. In other words, we have to show that such a function can be computed. From the proof of *(ex z being Any st Spec[f, x, z])*, we have a way to compute *(f x)* by using references to *f* itself. Since the only place that we can get reference to *f* is from the recursion hypothesis, and the recursion hypothesis is only useful for elements that are simpler than x, therefore all the references to *f* itself called by *(f x)* are applied to elements that are simpler than x. Moreover, [21] states the following:

The function f is defined by well-founded recursion over the relation $\prec$ if it has the form

$$(f\,a) \equiv_{df} \ldots (f\,a_1) \ldots (f\,a_n) \ldots$$

where each $a_i \prec a$.
Therefore, the *recursive definition* rule is correct. The above reasoning gives us not only a proof sketch of the *recursive definition* but also some insight on how to implement the realization of the rule. The realization of *recursive definition* is shown in Appendix A.

5.2 A Small Example

We want to illustrate the use of *recursive definition* by a small example. This should be a recursive function which is both easy to state as well as easy to prove. The identity function is a good choice.

We define the identity function of a bit string x by the concatenation of the first bit of x and the identity function of the rest bits of x. In order to simplify the proof, we derived the functions *first_bit* and *rest_bits* in MIZAR-C as follows:

```
given rec.rest_bits being (Bits -> Bits);

rec.rest_bits: (for x being Bits holds
   (ex f2 being Bits st
      ((split <x, 0>)=<f2, (rec.rest_bits x)>)));

given rec.first_bit being (Bits -> Bits);

rec.first_bit: (for x being Bits holds
   (ex r1 being Bits st
    ((split <x, 0>)=<(rec.first_bit x), r1>));

rec.first_rest: (for x being Bits holds
    ((split <x, 0>)=<(rec.first_bit x), (rec.rest_bits x)>));
```

The function *first_bit* returns the first bit of a bit string if there is one. Otherwise, it returns *nil*. On the other hand, the function *rest_bits* returns the remaining bit string once the first bit is removed. It returns *nil* if the input bit string is *nil*.

Before we prove anything, we have to state the specifications of the identity function formally as follows:

```
spec_def: define spec of f being (Bits -> Bits), x being Bits, z being Bits by
    ((z = z) & /* certificate of a total function */
    ((x = nil) implies (z = nil)) & /* termination case */
    ((x <> nil) implies
       (z = (cat <(first_bit x), (f (rest_bits x))>))) ) /* recursive definition */;
```

The proof of the identity function is shown below. The proof is kind of easy to follow. First, we prove *(ex z being Bits st spec[f,x,z])* for x=*nil*. This can be treated as the simplest case or the termination case of the recursion. Second, we prove *(ex z being Bits st spec[f,x,z])* for x<>*nil*. This is where the actual recursion takes place. Finally, we summarize the two cases by *case analysis* and then use *recursive definition* to introduce the identity function.

```
now
  /* Proof of the recursive identity function */
  let x be Bits;

  now
    let f be (Bits -> Bits);

    /* Recursion Hypothesis */
    assume RecHyp: (for y being Bits holds
       (LT[y, x] implies spec[f, y, (f y)])
    );
    cases_x_nil_or_not: ( (x = nil) or (x <> nil) ) by elim[x](BA_nil_or_not);
    x_nil_case: now
       /* When x is nil. */
       assume x_nil: (x = nil);
       conj1: (x = x) by eqintro();
       ( (x <> nil) or (x = nil) ) by disjintro(x_nil);
       conj2: ((x = nil) implies (x = nil)) by disj2imp(_PREVIOUS);
       ( (x = nil) or (x = (cat <(first_bit x), (f (rest_bits x))>)) ) by disjintro(x_nil);
       conj3: ((x <> nil) implies (x = (cat <(first_bit x), (f (rest_bits x))>)) )
          by disj2imp(_PREVIOUS);
       ( (x = x) & ((x = nil) implies (x = nil)) & ((x <> nil) implies
      (x = (cat <(first_bit x), (f (rest_bits x)) >) )) ) by conj(conj1, conj2, conj3);
    spec[f, x, x ] by definition(spec_def, _PREVIOUS);

    /* the function returns nil */
    thus (ex z being Bits st
      spec[f, x, z ]
    ) by exintro(_PREVIOUS);

end;

x_not_nil_case: now
    /* When x is not nil, */
    assume x_not_nil: (x <> nil);
    first_bit_ref: ((split <x, 0 >) = <(first_bit x), (rest_bits x) >) by elim [x](first_rest);
    bits_len_lt[(rest_bits x), x ] by elim[x,0,(first_bit x),(rest_bits x) ]
      (7.3, _PREVIOUS, _PREVIOUS, x_not_nil, BA_nil_not_0, _PREVIOUS);

    /* (rest_bits x) is simpler than x */
    LT[(rest_bits x), x ] by elim[(rest_bits x),x](BA_LT, _PREVIOUS, _PREVIOUS );

    /* reference to (f (rest_bits x)) */
    spec[f, (rest_bits x), (f (rest_bits x)) ]
      by elim[(rest_bits x)](RecHyp, _PREVIOUS, _PREVIOUS);
    ( ((f (rest_bits x)) = (f (rest_bits x))) & (((rest_bits x) = nil) implies
      ((f (rest_bits x)) = nil)) & (((rest_bits x) <> nil) implies
      ((f (rest_bits x)) = (cat <(first_bit (rest_bits x)), (f (rest_bits (rest_bits x))) >))))
      by definition(spec_def, _PREVIOUS);
    consider z being Bits such that
    (z = (cat <(first_bit x), (f (rest_bits x) ) >))
      by elim[(first_bit x),(f (rest_bits x)) ](BA_cat, first_bit_ref, _PREVIOUS);
    ( (x = nil) or (z = (cat <(first_bit x), (f (rest_bits x)) >)) )
      by disjintro(_PREVIOUS);
    conj3: ((x <> nil) implies (z = (cat <(first_bit x), (f (rest_bits x)) >)) )
      by disj2imp(_PREVIOUS);
    conj1: (z = z) by eqintro();
    ( (x <> nil) or (z = nil) ) by disjintro(x_not_nil);
    conj2: ((x = nil) implies (z = nil)) by disj2imp(_PREVIOUS);
    ( (z = z) & ((x = nil) implies (z = nil)) & ((x <> nil) implies
      (z = (cat <(first_bit x), (f (rest_bits x)) >) )) ) by conj(conj1, conj2, conj3);
    spec[f, x, z ] by definition(spec_def, _PREVIOUS);

    /* the function returns (cat <(first_bit x), (f (rest_bits x)) >) */
    thus (ex zz being Bits st
      spec[f, x, zz ]
    ) by exintro(_PREVIOUS);
    /* Combine the 2 cases by case analysis */
    thus (ex z being Bits st
      spec[f, x, z ]
    ) by caseanal(cases_x_nil_or_not, x_nil_case, x_not_nil_case);

  end;

  thus (for f being (Bits -> Bits) holds
    ((for y being Bits holds
      (LT[y, x ] implies spec[f, y, (f y) ])
    ) implies (ex z being Bits st
      spec[f, x, z ]
    ))
  ) by direct(_PREVIOUS);

end;


/* Extract the function */
(ex id being (Bits -> Bits) st
  (for x being Bits holds
    spec[id, x, (id x) ]
  )
) by recdef(_PREVIOUS);
```

5.3 Recursive Definition Versus Choice

One interesting observation is that the *recursive definition* rule can actually replace the *choice* rule (Section 3.4) in introducing new functions. In other words, the *recursive definition* rule is not restricted to introduce recursive functions only. This is easy to understand because if we do not include the function itself to be part of the specifications, the resulting specifications would then be the specifications of a non-recursive function.

We illustrate this interesting fact by introducing the non-recursive identity function using first the *choice* rule and then the *recursive definition* rule; and then comparing the contents of the two proofs. After that, we compare their actual extracted Lisp code. The non-recursive identity function of a bit string *x* is simply defined as the bit string *x* itself.

The proof of the non-recursive identity function using the *choice* rule is shown below:

```
now
  let x be Bits;
```

```
    (x = x) by eqintro();
    thus (ex y being Bits st
      (y = x)
    ) by exintro(_PREVIOUS);

  end;


  (ex f being (Bits -> Bits) st
    (for x being Bits holds
      ((f x) = x)
    )
  ) by choice(_PREVIOUS);
```

The proof of the non-recursive identity function using the *recursive definition* rule is shown below:

```
  spec_def: define spec of f being (Bits -> Bits), x being Bits, z being Bits by (z = x);

  now
    let x be Bits;

    now
      let f be (Bits -> Bits);

      assume RecHyp: (for y being Bits holds
        (LT[y, x ] implies spec[f, y, (f y) ])
      );
      (x = x) by eqintro();
      (ex z being Bits st
        (z = x)
      ) by exintro(_PREVIOUS);
      thus (ex z being Bits st
        spec[f, x, z ]
      ) by definition(spec_def, _PREVIOUS);

    end;

    thus (for f being (Bits -> Bits) holds
      ((for y being Bits holds
        (LT[y, x ] implies spec[f, y, (f y) ])
      ) implies (ex z being Bits st
        spec[f, x, z ]
      ))
    ) by direct(_PREVIOUS);

  end;
```

```
(ex f being (Bits -> Bits) st
   (for x being Bits holds
      specif. x. (f x) )
   )
) by recdef(_PREVIOUS);
```

Although the second proof seems to be much longer and complicated, the actual contents of both proofs are basically the same. All the remaining details of the second proof are only the format of using the *recursive definition* rule.

```
(LIST (APPLY0 (LAMBDA NIL
   (LAMBDA (x)
      (APPLY0 (APPLY (LAMBDA (y) (LAMBDA NIL y))
         x)))
   ))
```

Figure 5.1: Lisp program extracted from the proof of the non-recursive identity function using *choice*

```
(LIST (APPLY0 (LAMBDA NIL
   (LAMBDA (x)
      (APPLY (REC-DEF (LAMBDA (y)
         (LAMBDA (z)
            (LAMBDA NIL z)))
               )
            x)))
         ))
```

Figure 5.2: Lisp program extracted from the proof of the non-recursive identity function using *recursive definition*

The Lisp programs extracted from the proof using *choice* and the proof using *recursive definition* are shown in Figure 5.1 and Figure 5.2 respectively. Refer to Appendix A for the definition of *REC-DEF* and *APPLY0*. After simplifying, the two programs both result in the following simple Lisp program:

```
(LIST (LAMBDA (x)
   x))
```

Chapter 6

Recursive Functions for Binary Trees

Once the *recursive definition* rule was defined, we could derive recursive functions in MIZAR-C. As our initial motivation was to derive the traversal function, our first recursive function derived for binary trees was the traversal function which gives the in-order traversal of a binary tree. Next, we derived the depth function which gives the depth of a binary tree expressed in unary naturals.

6.1 The Traversal Function

We derived the traversal function for binary trees in MIZAR-C as follows:

```
b3.trav_spec: define b3.trav_spec of f being (Bits -> Bits), x being Bits, z being Bits by
  ( (z=z) &
  ((not bt.btree[x]) implies (z=nil)) &
  (bt.btree[x] implies (z=(cat <(cat <(f (bt.l_tree x)), (bt.data x)>), (f (bt.r_tree x))>))) );

given b3.trav being (Bits -> Bits);

b3.trav: (for x being Bits holds
  b3.trav_spec[b3.trav, x, (b3.trav x)]);
```

The proof is shown in Appendix B.

We illustrate how the traversal function works by an example. Figures 6.1 to 6.4 show the step-by-step construction of a binary tree S_4 with four nodes m_1 to m_4. Table 6.1 shows the data stored in each of the nodes.

Figure 6.1 shows a binary tree S_1 which consists of the root m_1, an empty left sub-tree and an empty right sub-tree. Figure 6.2 shows a tree S_2 which consists of the root m_2, an empty left sub-tree and a right sub-tree S_1. Figure 6.3 shows a tree S_3 which consists of the root m_3, an empty left sub-tree and an empty right sub-tree. Finally, Figure 6.4 shows the binary tree S_4 which consists of the root m_4, a left sub-tree S_2 and a right sub-tree S_3.

Node	Data
m_1	*1*
m_2	*0*
m_3	*11*
m_4	*10*

Table 6.1: Data stored in nodes m_1 to m_4

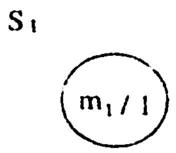

Figure 6.1: Binary tree S_1

We want to show how *(trav S4)* is computed. We denote the concatenation of two bit strings x and y by x·y. From the specifications of the traversal function, we have:

$$(trav\ S4) = (trav\ S2) \cdot 00110 \cdot (trav\ S3).$$

Similarly, we have:

$$\begin{aligned}(trav\ S2) &= (trav\ nil) \cdot 010 \cdot (trav\ S1).\\ (trav\ nil) &= nil.\\ (trav\ S1) &= (trav\ nil) \cdot 011 \cdot (trav\ nil).\\ (trav\ S3) &= (trav\ nil) \cdot 00111 \cdot (trav\ nil).\end{aligned}$$

Therefore, the computed results are:

$$\begin{aligned}(trav\ S1) &= 011\\ (trav\ S2) &= 010011\\ (trav\ S3) &= 00111\\ (trav\ S4) &= 0100110011000111\end{aligned}$$

6.2 The Depth Function

The definition of the depth function of a binary tree is recursively defined as follows:

- the depth of an empty tree is 0 (decimal).
- the depth of a non-empty binary tree *x* is 1 (decimal) plus the maximum of the depth of the left sub-tree of *x* and the depth of the right sub-tree of *x*

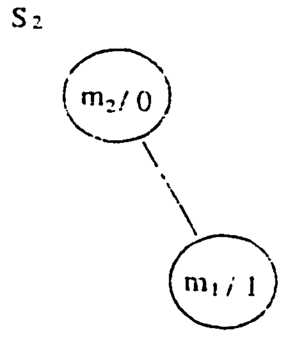

Figure 6.2: Binary tree S_2

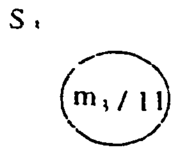

Figure 6.3: Binary tree S_3

As we have to represent the depth of a binary tree in a bit string, we express the depth in unary naturals. Before we derived the depth function, we had to derive the function *max* which takes in two individual bit strings and returns the length of the bit string with length no less than the other. For example:

((max 0) 00) = 00
((max 101) 011) = 000
((max nil) 00) = 00
((max nil) nil) = nil

```
given b4.max being (Bits -> (Bits -> Bits)):

b4.max: (for x being Bits holds
   (for y being Bits holds
      ( (((b4.max x) y)=((b4.max x) y)) &
      (bits_len_lt[x, y] implies (((b4.max x) y)=( U.length y))) &
      ((not bits_len_lt[x, y]) implies (((b4.max x) y)=( U.length x))) )));
```

We derived the depth function for binary trees as follows:

```
b4.depth_spec: define b4.depth_spec of f being (Bits -> Bits), x being Bits, z being Bits by
```

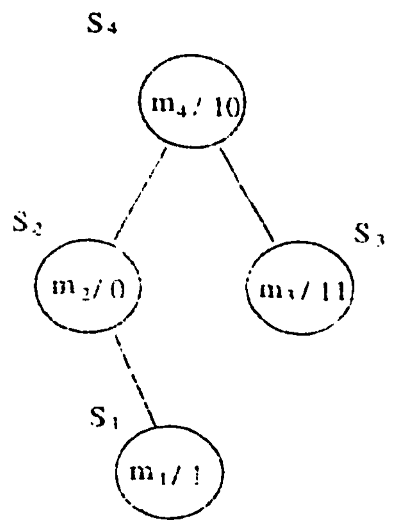

Figure 6.4: Binary tree S_4

```
    ! (z=z) &
    ((not bt.btree[x]) implies (z=nil)) &
    (bt.btree[x] implies (z=(cat <((b4.max (f (bt.l_tree x))) (f (bt.r_tree x))), 0>))) );

  given b4.depth being (Bits -> Bits);

  b4.depth: (for x being Bits holds
    b4.depth_spec[b4.depth, x, (b4.depth x)]);
```

The proof is shown in Appendix C.

We illustrate how the depth function works by computing the depth of the binary tree S_4 (Figure 6.4). From the specifications of the function, we have:

(depth S4) = (max (depth S2) (depth S3)) · 0.

Similarly, we have:

(depth S2) = (max (depth nil) (depth S1)) · 0.
(depth nil) = nil.
(depth S1) = (max (depth nil) (depth nil)) · 0.
(depth S3) = (max (depth nil) (depth nil)) · 0.

Therefore, the computed results are:

(depth S1) = 0,
(depth S2) = 00,
(depth S3) = 0,
(depth S4) = 000.

6.3 Complexity Analysis

Being able to derive recursive functions is not enough, we also want to have an idea on how these functions perform. In this section, we compute the time complexities of the traversal and the depth functions.

First, let us look at the time complexity of the traversal function. Let n be the size and d be the depth of the input binary tree. The first call to the traversal function requires $O(n)$ time, since the whole tree has to be scanned through. The function will then call two copies of itself and each of them will take in a bit string with length less than $n/2$. This is because every sub-tree is either *nil* or wrapped in a *Packet*. The length of that sub-tree is less than half the length of its *Packet*, and so is less than half the length of the original tree. This process continues until the whole tree is traversed.

Level of recursion	Time	Maximum number of copies of the function running
1	$O(n)$	1
2	$O(n/2)$	2
3	$O(n/4)$	4
$\vdots$	$\vdots$	$\vdots$
d	$O(1)$	n

Table 6.2: Time complexity of *traversal*

Table 6.2 shows the time required and the maximum number of copies of the function running for each level of recursion. Summing each level up gives:

$$\begin{aligned} & \underbrace{O(n) \times 1 + O(n/2) \times 2 + O(n/4) \times 4 + \cdots + O(1) \times n}_{d \text{ times}} \\ = \quad & O(d \cdot n) \end{aligned}$$

Therefore, the time complexity of the *traversal* function that we derived is $O(d \cdot n)$ where d and n are the depth and the size of the binary tree respectively.

Similarly, the time complexity of the *depth* function is the same as that of the *traversal* function. This is due to the fact that the *max* function is a linear function and so the time complexity of computing

$$((max\ (depth\ (l\_tree\ x)))\ (depth\ (r\_tree\ x)))$$

is the same as that of computing both

$$(depth\ (l\_tree\ x)) \text{ and } (depth\ (r\_tree\ x)).$$

Hence, using similar reasoning as we do in computing the time complexity of the *traversal* function, the time complexity of the *depth* function is also $O(d \cdot n)$ where d and n are the depth and the size of the binary tree respectively.

Chapter 7

Conclusions

7.1 Ordered Binary Trees

The binary tree is not an efficient data structure for search unless the data stored in each node is ordered. We call such a binary tree an ordered binary tree. In order to access the data efficiently, we need to have some operations on ordered binary trees. The most basic operations needed are search, insertion, and deletion. If we are to use ordered binary trees as a data structure in MIZAR-C, we have to be able to perform these operations in MIZAR-C efficiently.

Search can be easily implemented with our existing representation of binary trees. As binary trees are represented as a finite sequence with 3 *Packets* in MIZAR-C, a search is simply a comparison of the search item with the data stored in the first *Packet* and then a search on the appropriate sub-tree represented by either the second or the third *Packet*. The search ends when the item is found or an empty sub-tree is reached.

What is the time complexity of search? The first call requires $O(n)$ as the whole binary tree has to be scanned through. The time required for each subsequent call is at least reduced to half of that of the previous call. Therefore, the time complexity of search is

$$\begin{aligned} & O(n) + O(n/2) + O(n/4) + \cdots + O(1) \\ = \; & O(2n) \\ = \; & O(n) \end{aligned}$$

Hence, it is linear.

On the other hand, insertion and deletion cannot be done as easily as search. An insertion normally means a search for the appropriate location and then an addition of a new leaf to the binary tree. As binary trees are represented as "*Packets* in *Packets*" in MIZAR-C, an addition of a leaf requires a change of the content of the innermost *Packet* and thus requires changes of the contents of all related outer *Packets*. Deletion is even worse. A deletion normally requires re-location of sub-trees which is a very difficult task with our existing representation of binary trees. So, our next step in developing binary trees as a data structure is to efficiently implement insertion and deletion.

7.2 Proof Checking Environment

In MIZAR-C, and many other formal proof checking environments, the users often have to deal with a large amount of details. In order to increase the productivity, the proof checking environment needs to be improved.

One suggestion to improve the proof checking environment in MIZAR-C is to increase the automatic theorem proving power of the system. Often the users have to spend a lot of time proving something that contributes nothing to the computations at all. If we can let the system handle these details, the users can have more time to concentrate on the "meaningful" parts of the proofs. However, this automatic help cannot be done in the constructive part of the proof without influencing the resulting code. After all, it is the contents of the proof that gives us the program.

When proving a more complicated statement, the user usually needs to include a large number of references. Moreover, these references need to be entered in sequence. If the system can prompt the user for the correct type of reference, errors made by the user can be greatly reduced.

Another way to reduce the time the users need to spend on writing proofs is to have the system provide an outline of the proof. So, all the users are responsible to do are to fill in the details of the outline. This is possible because many inference rules have their own formats. For example, *case analysis* has the following general format:

```
cases: for x being Any holds P[x] or not P[x];

P_case: now
  assume P[x];
  ⋮
  thus Q[x];
end;

not_P_case: now
  assume (not P[x]);
  ⋮
  thus Q[x];
end;

thus Q[x] by caseanal(cases, P_case, not_P_case);
```

Other inference rules such as *induction* and *recursive definition* have their own formats, too.

A large proof usually contains many sub-proofs, or sub-theorems. Once a sub-theorem has been proven, it is not necessary for the user to see the details of the sub-proof again as it is impossible to reference any statement within the scope of the sub-theorem from outside anyways. Since all the user needs is the sub-theorem statement, the system can hide the rest of the details unless the user wants to refer back to them. This can provide a more high-level proof outline to the user.

Appendix A

Realization of Recursive Definition

In general, a *recursive definition* looks like:

```
rdstep: (for x being Any holds
   (for f being (Any -> Any) hold
      ((for y being Any holds
         (LT[y, x ] implies spec[f, y, (f y) ])
      ) implies (ex z being Any st
         spec[f, x, z ]
      ))
   );
) ;

(ex f being (Any -> Any) st
   (for x being Any holds
      spec[f, x, (f x) ]
   )
) by recdef(rdstep);
```

The realization of *recursion definition* looks like:

```
(defun REC-DEF (rdstep)

   ; return LAMBDA form that computes f
   (LAMBDA (x)
      (APPLY0
         (APPLY
            (APPLY rdstep x)
            (APPLY REC-DEF rdstep)
         )
      )
   )
) ; end defun
```

```
(DEFUN APPLY0 (func)
  (APPLY func NIL)
)
```

Figure A.1: Definition of the function APPLY0

where *APPLY0* is defined as the function unwrapping the *LAMBDA* expression (Figure A.1).

REC-DEF takes in the *recursive definition* proof and returns the *LAMBDA* form that computes f. It works by applying the proof to x and then f. Finally, it unwraps the resulting *LAMBDA* expression and wraps it up again in the *LAMBDA* form. In fact, we got the idea on how to implement the realization of *recursive definition* from that of *induction*.

In general, an *induction* looks like:

```
istep: (for x being (Any -> Any) holds
  ((for y being Any holds
    (LT[y, x ] implies P[y])
  ) implies P[x])
);

(for x being Any holds
  P[x]) by induction(istep);
```

The realization of *induction* looks like:

```
(defun GEN-IND (istep)

  ; return LAMBDA form that computes (for x holds P[x])
  (LAMBDA (x)
    (APPLY
      (APPLY istep x) ;; needs to be applied to IH
      (LAMBDA (y) (APPLY (GEN-IND istep) y))
    )
  )
)
```

GEN-IND takes in the *induction* proof, which has type

```
x -> (IH -> P[x])
```

and returns the *LAMBDA* form that computes *(for x holds P[x])*. It works by applying the proof to x and then the induction hypothesis, which has type

```
y -> (LT[y,x] -> P[y]).
```

Finally, it wraps the result up in the *LAMBDA* form.

Appendix B

Proof of the Traversal Function

```
/* Specifications of traversal */
trav_spec: define trav_spec of f being (Bits -> Bits), x being Bits, z being Bits by
  ( (z = z) &
  ((not bt.btree[x ]) implies (z = nil)) &
  (bt.btree[x ] implies (z=(cat <(cat <(f (bt.l_tree x)), (bt.data x) >), (f (bt.r_tree x)) >))));

now
  /* Proof of traversal */
  let x be Bits;

  now
    let f be (Bits -> Bits);

    /* Recursion Hypothesis */
    assume RD: (for y being Bits holds
      (LT[y, x ] implies trav_spec[f, y, (f y) ])
    );
    cases_btree_or_not: ( bt.btree[x ] or (not bt.btree[x ]) ) by elim[x](b2.btree_decide);
    btree_x_case: now
      /* When x is a binary tree. */
      assume btree_x: bt.btree[x ];
      x_not_nil: (x <> nil) by elim[x](b2.btree_not_nil._PREVIOUS);
      bits_len_lt[(bt.l_tree x), x ] by elim[x](b2.l_tree_lt_x, _PREVIOUS);

      /* (l_tree x) is simpler than x */
      LT[(bt.l_tree x), x ] by elim[(bt.l_tree x),x](BA_LT, _PREVIOUS, _PREVIOUS);

      /* reference to (f (bt.l_tree x)) */
      trav_spec[f, (bt.l_tree x), (f (bt.l_tree x)) ]
        by elim[(bt.l_tree x)](RD, _PREVIOUS, _PREVIOUS);
      1st: ( ((f (bt.l_tree x)) = (f (bt.l_tree x))) & ((not bt.btree[(bt.l_tree x) ]) implies
        ((f (bt.l_tree x)) = nil)) & (bt.btree[(bt.l_tree x) ] implies ((f (bt.l_tree x)) =
        (cat <(cat <(f (bt.l_tree (bt.l_tree x))), (bt.data (bt.l_tree x)) >),
      (f (bt.r_tree (bt.l_tree x))) >))) ) by definition(trav_spec, _PREVIOUS);
    ((bt.data x) = (fseq.first x)) by elim[x](bt.data);
    consider z being Bits such that
    z_def: (z = (cat <(f (bt.l_tree x)), (bt.data x) >))
      by elim[(f (bt.l_tree x)),(bt.data x) ](BA_cat, 1st, _PREVIOUS);
    bits_len_lt[(bt.r_tree x), x ] by elim[x](b2.r_tree_lt_x, x_not_nil);

    /* (bt.r_tree x) is simpler than x */
    LT[(bt.r_tree x), x ] by elim[(bt.r_tree x),x](BA_LT, _PREVIOUS, _PREVIOUS);

    /* reference to (f (bt.r_tree x)) */
    trav_spec[f, (bt.r_tree x), (f (bt.r_tree x)) ]
      by elim[(bt.r_tree x)](RD, _PREVIOUS, _PREVIOUS);
    ( ((f (bt.r_tree x)) = (f (bt.r_tree x))) & ((not bt.btree[(bt.r_tree x) ]) implies
      ((f (bt.r_tree x)) = nil)) & (bt.btree[(bt.r_tree x) ] implies ((f (bt.r_tree x)) =
      (cat <(cat <(f (bt.l_tree (bt.r_tree x))), (bt.data (bt.r_tree x)) >),
      (f (bt.r_tree (bt.r_tree x))) >))) ) by definition(trav_spec, _PREVIOUS);
    consider z' being Bits such that
    (z' = (cat <z, (f (bt.r_tree x)) >)) by elim[z,(f (bt.r_tree x))](BA_cat, _PREVIOUS);
    (z' = (cat <(cat <(f (bt.l_tree x)), (bt.data x) >), (f (bt.r_tree x)) >))
      by equality(_PREVIOUS, z_def);
    ( (not bt.btree[x ]) or (z' = (cat <(cat <(f (bt.l_tree x)), (bt.data x) >),
      (f (bt.r_tree x)) >)) ) by disjintro(_PREVIOUS);
    conj3: (bt.btree[x ] implies (z' = (cat <(cat <(f (bt.l_tree x)), (bt.data x) >),
      (f (bt.r_tree x)) >) )) by disj2imp(_PREVIOUS);
    conj1: (z' = z') by eqintro();
    ( bt.btree[x ] or (z' = nil) ) by disjintro(btree_x);
    ((not bt.btree[x ]) implies (z' = nil)) by disj2imp(_PREVIOUS),
    ( (z' = z') & ((not bt.btree[x ]) implies (z' = nil)) & (bt.btree[x ] implies
      (z' = (cat <(cat <(f (bt.l_tree x)) , (bt.data x) >), (f (bt.r_tree x)) >))) )
      by conj(conj1, _PREVIOUS, conj3);
    trav_spec[f, x, z' ] by definition(trav_spec, _PREVIOUS);

    /* it returns (cat <(cat <(f (bt.l_tree x)), (bt.data x) >), (f (bt.r_tree x)) >) */
    thus (ex zz being Bits st
      trav_spec[f, x, zz ]
    ) by exintro(_PREVIOUS);

end;

not_btree_x_case: now
    /* When x is not a binary tree. */
    assume not_btree_x: (not bt.btree[x ]);
    conj1: (nil = nil) by eqintro();
    ( bt.btree[x ] or (nil = nil) ) by disjintro(_PREVIOUS);
    conj2: ((not bt.btree[x ]) implies (nil = nil)) by disj2imp(_PREVIOUS);
```

```
        ( (not bt.btree[x ]) or (nil = (cat <(cat <(f (bt.l_tree x)), (bt.data x) >),
           (f (bt.r_tree x)) >) ) ) by disjintro(not_btree_x);
        (bt.btree[x ] implies (nil = (cat <(cat <(f (bt.l_tree x)), (bt.data x) >),
           (f (bt.r_tree x)) >))) by disj2imp(_PREVIOUS);
        ( (nil = nil) & ((not bt.btree[x ]) implies (nil = nil)) & (bt.btree[x ] implies
           (nil = (cat <(cat <(f (bt.l_tree x)) , (bt.data x) >), (f (bt.r_tree x)) >))) )
           by conj(conj1, conj2, _PREVIOUS);
        trav_spec[f, x, nil ] by definition(trav_spec, _PREVIOUS);

        /* it returns nil */
        thus (ex zz being Bits st
           trav_spec[f, x, zz ]
        ) by exintro(_PREVIOUS);

     end;

     /* Combine the 2 cases by case analysis */
     thus (ex z being Bits st
        trav_spec[f, x, z ]
     ) by caseanal(cases_btree_or_not, btree_x_case, _PREVIOUS);

  end;

  thus (for f being (Bits -> Bits) holds
     ((for y being Bits holds
        (LT[y, x ] implies trav_spec[f, y, (f y) ])
     ) implies (ex z being Bits st
        trav_spec[f, x, z ]
     ))
  ) by direct(_PREVIOUS);

end;


/* Re-stating the specifications */
(for x being Bits holds
  (for f being (Bits -> Bits) holds
     ((for y being Bits holds
        (LT[y, x ] implies trav_spec[f, y, (f y) ])
     ) implies (ex z being Bits st
        trav_spec[f, x, z ]
     ))
  )
) by direct(_PREVIOUS);

/* Extract the function */
```

```
(ex f being (Bits -> Bits) st
   (for x being Bits holds
      trav_spec(f, x, (f x))
   )
) by recdef(_PREVIOUS);

/* Name the function */
consider trav being (Bits -> Bits) such that trav: (for x being Bits holds
   trav_spec(trav, x, (trav x))
) by direct(_PREVIOUS);
```

Appendix C

Proof of the Depth Function

```
/* Proof of max */
now
  /* x is the first input bit string */
  let x be Bits;

  now
    /* y is the second input bit string */
    let y be Bits;

    cases: ( bits_len_lt[x, y] or (not bits_len_lt[x, y])) by elim[x,y](lt_decide.len_lt_or_not);
    lt: now
      /* When x is shorter than y, */
      assume lt: bits_len_lt[x, y ];
      bits_len_eq[y, y ] by elim[y](BA_len_eq_refl);
      conj1: ((U.length y) = (U.length y)) by elim[y,y](U.unat2, _PREVIOUS);
      ((not bits_len_lt[x, y ]) or ((U.length y)=(U.length y))) by disjintro(_PREVIOUS);
      conj2: (bits_len_lt[x, y ] implies ((U.length y) = (U.length y)))
        by disj2imp(_PREVIOUS);
      ( bits_len_lt[x, y ] or ((U.length y) = (U.length x)) ) by disjintro(lt);
      ((not bits_len_lt[x, y ]) implies ((U.length y) = (U.length x)))
        by disj2imp(_PREVIOUS);
      ( ((U.length y) = (U.length y)) & (bits_len_lt[x, y ] implies ((U.length y) =
        (U.length y))) & ((not bits_len_lt[x, y ]) implies ((U.length y) = (U.length x))) )
        by conj(conj1, conj2, _PREVIOUS);

      /* it returns the length of y */
      thus (ex z being Bits st
        ( (z = z) & (bits_len_lt[x, y ] implies (z = (U.length y))) &
        ((not bits_len_lt[x, y ]) implies (z = (U.length x))) )
      ) by exintro(_PREVIOUS, _PREVIOUS);

    end;
    not_lt: now
      /* When x is no shorter than y. */
      assume not_lt: (not bits_len_lt[x, y ]);
      bits_len_eq[x, x ] by elim[x]( BA_len_eq_refl);
      conj1: ((U.length x) = (U.length x)) by elim[x,x]( U.unat2, _PREVIOUS);
      ( (not bits_len_lt[x, y ]) or ((U.length x) = (U.length y)) ) by disjintro(not_lt);
      conj2: (bits_len_lt[x, y ] implies ((U.length x) = (U.length y)))
         by disj2imp(_PREVIOUS);
      ( bits_len_lt[x, y ] or ((U.length x) = (U.length x)) ) by disjintro(conj1);
      ((not bits_len_lt[x, y ]) implies ((U.length x) = (U.length x)))
         by disj2imp(_PREVIOUS);
      ( ((U.length x) = (U.length x)) & (bits_len_lt[x, y ] implies ((U.length x) =
         (U.length y))) & ((not bits_len_lt[x, y ]) implies ((U.length x) = (U.length x))) )
         by conj(conj1, conj2, _PREVIOUS);

      /* it returns the length of x */
      thus (ex z being Bits st
         ( (z = z) & (bits_len_lt[x, y ] implies (z = (U.length y))) &
         ((not bits_len_lt[x, y ]) implies (z = (U.length x))) )
      ) by exintro(_PREVIOUS, _PREVIOUS);

    end;

    /* Combine the 2 cases by case analysis */
    thus (ex z being Bits st
       ( (z = z) & (bits_len_lt[x, y ] implies (z = (U.length y))) &
       ((not bits_len_lt[x, y ]) implies (z = (U.length x))) )
    ) by caseanal(cases, lt, _PREVIOUS);

  end;

  thus (ex f being (Bits -> Bits) st
    (for y being Bits holds
       ( ((f y) = (f y)) & (bits_len_lt[x, y ] implies ((f y) = (U.length y))) &
       ((not bits_len_lt[x, y ]) implies ((f y) = (U.length x))) )
    )
  ) by choice(_PREVIOUS);

end;


/* Use choice twice to extract the function as it has 2 inputs */
(ex max being (Bits -> (Bits -> Bits)) st
  (for x being Bits holds
    (for y being Bits holds
       ( (((max x) y) = ((max x) y)) &
        (bits_len_lt[x, y ] implies (((max x) y) = (U.length y))) &
        ((not bits_len_lt[x, y ]) implies (((max x) y) = (U.length x))) )
      )
    )
  ) by choice(_PREVIOUS);

  /* Name the function */
  consider max being (Bits -> (Bits -> Bits)) such that max: (for x being Bits holds
    (for y being Bits holds
      ( (((max x) y) = ((max x) y)) &
      (bits_len_lt[x, y ] implies (((max x) y) = (U.length y))) &
      ((not bits_len_lt[x, y ]) implies (((max x) y) = (U.length x))) )
    )
  ) by direct(_PREVIOUS);

  /* Specifications of depth */
  depth_spec: define depth_spec of f being (Bits -> Bits), x being Bits, z being Bits by
    ( (z = z) &
    ((not bt.btree[x ]) implies (z = nil)) &
    (bt.btree[x ] implies (z = (cat <(((max (f (bt.l_tree x))) (f (bt.r_tree x))) . 0 >))) );

    now
      /* Proof of depth */
      let x be Bits;

      now
        let f be (Bits -> Bits);

        /* Recursion Hypothesis */
        assume RD: (for y being Bits holds
          (LT[y, x ] implies depth_spec[f, y, (f y) ])
        );
        cases_btree_or_not: ( bt.btree[x ] or (not bt.btree[x ]) ) by elim[x](b2.btree_decide);
        btree_case: now
          /* When x is a binary tree. */
          assume btree: bt.btree[x ];
          x_not_nil: (x <> nil) by elim[x](b2:btree_not_nil, _PREVIOUS);
          bits_len_lt[(bt.l_tree x), x ] by elim[x](b2.l_tree_lt_x, _PREVIOUS);

          /* (bt.l_tree x) is simpler than x */
          LT[(bt.l_tree x), x ] by elim[(bt.l_tree x),x](BA_LT, _PREVIOUS, _PREVIOUS);

          /* reference to (f (bt.l_tree x)) */
          depth_spec[f, (bt.l_tree x), (f (bt.l_tree x)) ]
            by elim[(bt.l_tree x)](RD, _PREVIOUS, _PREVIOUS);
          l_tree_ref: ( ((f (bt.l_tree x)) = (f (bt.l_tree x))) & ((not bt.btree[(bt.l_tree x) ])
      implies ((f (bt.l_tree x)) = nil)) & (bt.btree[(bt.l_tree x) ] implies
      ((f (bt.l_tree x)) = (cat <((max (f (bt.l_tree (bt.l_tree x))))
      (f (bt.r_tree (bt.l_tree x)))), 0 >))) ) by definition(depth_spec, _PREVIOUS);
   bits_len_lt[(bt.r_tree x), x ] by elim[x](b2.r_tree_lt_x, x_not_nil);

   /* (bt.r_tree x) is simpler than x */
   LT[(bt.r_tree x), x ] by elim[(bt.r_tree x),x](BA_LT, _PREVIOUS, _PREVIOUS);

   /* reference to (f (bt.r_tree x)) */
   depth_spec[f, (bt.r_tree x), (f (bt.r_tree x)) ]
      by elim[(bt.r_tree x)](RD, _PREVIOUS, _PREVIOUS);
   r_tree_ref: ( ((f (bt.r_tree x)) = (f (bt.r_tree x))) & ((not bt.btree[(bt.r_tree x) ])
      implies ((f (bt.r_tree x)) = nil)) & (bt.btree[(bt.r_tree x) ] implies
      ((f (bt.r_tree x)) = (cat <((max (f (bt.l_tree (bt.r_tree x))))
      (f (bt.r_tree (bt.r_tree x)))), 0 >))) ) by definition(depth_spec, _PREVIOUS);
   (((max (f (bt.l_tree x))) (f (bt.r_tree x)))
   = ((max (f (bt.l_tree x))) (f (bt.r_tree x))))
      by elim[(f (bt.l_tree x)) ,(f (bt.r_tree x))](max, l_tree_ref, _PREVIOUS);
   consider z being Bits such that
   (z = (cat <((max (f (bt.l_tree x))) (f (bt.r_tree x))), 0 >))
      by elim[((max (f (bt.l_tree x)) ) (f (bt.r_tree x))),0](BA_cat, _PREVIOUS);
   ( (not bt.btree[x ]) or (z = (cat <((max (f (bt.l_tree x))) (f (bt.r_tree x))), 0 >)) )
      by disjintro(_PREVIOUS);
   conj3: (bt.btree[x ] implies (z = (cat <((max (f (bt.l_tree x))) (f (bt.r_tree x))),
      0 >))) by disj2imp(_PREVIOUS);
   ( bt.btree[x ] or (z = nil) ) by disjintro(btree);
   conj2: ((not bt.btree[x ]) implies (z = nil)) by disj2imp(_PREVIOUS);
   (z = z) by eqintro();
   ( (z = z) & ((not bt.btree[x ]) implies (z = nil)) & (bt.btree[x ] implies
      (z = (cat <((max (f (bt.l_tree x))) (f (bt.r_tree x))), 0 >))) )
      by conj(_PREVIOUS, conj2, conj3);
   depth_spec[f, x, z ] by definition(depth_spec, _PREVIOUS);

   /* it returns (cat <((max (f (bt.l_tree x))) (f (bt.r_tree x))), 0 >) */
   thus (ex z' being Bits st
      depth_spec[f, x, z' ]
   ) by exintro(_PREVIOUS);

end;

not_btree_case: now
   /* When x is not a binary tree, */
   assume not_btree: (not bt.btree[x ]);
   conj1: (nil = nil) by eqintro();
   ( bt.btree[x ] or (nil = nil) ) by disjintro(_PREVIOUS);
   conj2: ((not bt.btree[x ]) implies (nil = nil)) by disj2imp(_PREVIOUS);
```

```
        ((not bt.btree[x ]) or (nil = (cat <((max (f (bt.l_tree x))) (f (bt.r_tree x))), 0 >)))
          by disjintro(not_btree);
        (bt.btree[x ] implies (nil = (cat <((max (f (bt.l_tree x))) (f (bt.r_tree x))), 0 >)))
          by disj2imp(_PREVIOUS);
        ( (nil = nil) & ((not bt.btree[x ]) implies (nil = nil)) & (bt.btree[x ] implies
          (nil = (cat <((max (f (bt.l_tree x))) (f (bt.r_tree x))), 0 >))) )
          by conj(conj1, conj2, _PREVIOUS);
        depth_spec[f, x, nil ] by definition(depth_spec, _PREVIOUS);

        /* it returns nil */
        thus (ex z' being Bits st
          depth_spec[f, x, z' ]
        ) by exintro(_PREVIOUS);

      end;

      /* Combine the 2 cases by case analysis */
      thus (ex z' being Bits st
        depth_spec[f, x, z' ]
      ) by caseanal(cases_btree_or_not, btree_case, _PREVIOUS);

    end;

    thus (for f being (Bits -> Bits) holds
      ((for y being Bits holds
        (LT[y, x ] implies depth_spec[f, y, (f y) ])
      ) implies (ex z being Bits st
        depth_spec[f, x, z ]
      ))
    ) by direct(_PREVIOUS);

  end;


  /* Extract the function */
  (ex f being (Bits -> Bits) st
    (for x being Bits holds
      depth_spec[f, x, (f x) ]
    )
  ) by recdef(_PREVIOUS);

  /* Name the function */
  consider depth being (Bits -> Bits) such that depth: (for x being Bits holds
    depth_spec[depth, x, (depth x) ]
  ) by direct(_PREVIOUS);
```

Bibliography

[1] Michael J. Beeson. *Foundations of Constructive Mathematics*, volume 6 of *Ergebnisse der Mathematik und ihrer Grenzgebiete 3.Folge*. Springer-Verlag, Berlin-Heidelberg-New York, 1985.

[2] R. L. Constable and N. P. Mendler. Recursive definitions in type theory. Technical Report 85-659, Cornell University, January 1985.

[3] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, and S.F. Smith. *Implementing Mathematics with the* NUPRL *Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[4] Thierry Coquand. On the analogy between propositions and types. In Gerard Huet, editor, *Logical Foundations of Functional Programming*, The UT Year of Programming Series, pages 399–418, Reading, Massachusetts, 1990. Addison Wesley.

[5] G. Dowek et al. *The* COQ *Proof Assistant User's Guide*, February 1992.

[6] J. A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990.

[7] Susumu Hayashi. An introduction to PX. In Gerard Huet, editor, *Logical Foundations of Functional Programming*, The UT Year of Programming Series, pages 431–486. Reading, Massachusetts, 1990. Addison-Wesley.

[8] Susumu Hayashi and Hiroshi Nakano. PX*: A Computational Logic*. The MIT Press, Cambridge, Masschussets, 1989.

[9] H. J. Hoover and P. Rudnicki. *Introduction to Logic in Computing Science*. University of Alberta, Department of Computing Science, 1993.

[10] K. Kippen. Implementing bit data structures in MIZAR-C. Master's thesis, University of Alberta, 1995.

[11] S.C. Kleene. *Introduction to Metamathematics*. Van Nostrand, 1952.

[12] B. Knight. Safe strict evaluation of redundancy-free programs from proofs. Master's thesis, University of Victoria, 1994.

[13] Z. Manna and R. Waldinger. Fundamentals of deductive program synthesis. *IEEE Transactions on Sofware Engineering*, 18(8):674–704, August 1992.

[14] P. Martin-Lof. Constructive mathematics and computer programming. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 167–184. Prentice-Hall, 1985.

[15] C. J. Nix and B. P. Collins. The use of software engineering, including the Z notation, in the development of CICS. *Quality Assurance*, pages 103–110, September 1988.

[16] Bengt Nordstrom, Kent Peterson, and Jan M. Smith. *Programming in Martin-Lof Type theory*, volume 7 of *International Series of Monographs on Computer Science*. Claredon Press, Oxford, 1990.

[17] Frank Pfenning. Elf: A language for logic definition and verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322. IEEE Computer Society Press, June 1989.

[18] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator*. Texts and Monographs in Computer Science. Springer-Verlag, Berlin-Heidelberg-New York, 1989.

[19] Piotr Rudnicki and Wlodzimierz Drabent. Proving properties of pascal programs in MIZAR 2. *Acta Informatica*, 22:311–331, 1985.

[20] Robert Sedgewick. *Algorithms*. Addison-Wesley, second edition, 1988.

[21] Simon Thompson. *Type Theory and Functional Programming*. Addison-Wesley, 1991.

[22] A.S. Troelstra and D. van Dalen. *Constructivism in Mathematics: An Introduction, vols I and II*, volume 121 & 123 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1988.

[23] A. Walenstein, J. H. Hoover, and P. Rudnicki. Programming with constructive proof in the MIZAR-C proof environment. Technical Report 92-12, University of Alberta, 1992.

[24] J. M. Wing. A specifer's introduction to formal methods. *IEEE Computer*, 23(9):8–24, September 1990.