

# TECH REPORT: Trees in Constructive Type Theory

Josef Pohl

Department of Computer Science  
University of Wyoming

January 25, 2006

## 1 Introduction

Trees are pervasive in the literature of data structures and algorithms. Every computer science undergraduate has an understanding of at least how trees function as part of their chosen field as well as their formation. The foundations of many abstract data types rely on trees and the ability to quickly and efficiently search and organize large data sets often seems to be wholly dependent on trees.

In the current formal methods and theorem proving literature one can find instances where trees have been developed for specific purposes. In particular they are used to prove a few properties on a variety of data structures. In the online libraries of formal knowledge [1][2] a few small theories of trees, usually binary trees have been developed. It is rare to see one which formulates more than a few properties explicitly about trees. These formalizations often develop the appropriate constructors and destructors and give an abstract notion of a tree. This is where it usually stops. [5] Very rarely is a deeper investigation undertaken to look at the algorithmic and structural implications of a specific formalization. We begin to remedy this situation here.

In computer science and mathematics there are any number of ideas about what a tree is or how one would construct or represent a tree. A programmer familiar with C may think of a tree as a struct.

```
struct tree {  
    int a;  
    tree * left;  
    tree * right;  
}
```

It is practical and efficient, but it is hard to say much about it or what it means. A functional programmer (in ML for instance) may give a similar tree as

```
type 'a btree = Empty | Node of ('a * 'a btree * 'a btree);;
```

A functional representation is easier to reason about since it is closer to the logic we use to talk about the syntax and semantics of trees. The greatest breadth of tree characterizations comes from the graph theory definition of a tree. If in fact we regard trees as a class of constrained graphs we can come up with a number of different formulations. Berge gives us the following six characterizations of trees and shows that they are equivalent. [4]

**Theorem 1.1 (Berge’s equivalent tree representations)** *Let  $H$  be a graph with  $n$  nodes,  $n > 1$ ; any one of the following properties characterizes a tree:*

1.  *$H$  is connected and does not possess any cycles*
2.  *$H$  contains no cycles and has  $n - 1$  edges*
3.  *$H$  is connected and has  $n - 1$  edges*
4.  *$H$  contains no cycles, and if an edge is added which joins two non-adjacent vertices, one (and only one) cycle is thereby formed*
5.  *$H$  is connected but loses this property if any edge is deleted*
6. *every pair of vertices is connected by one and only one chain (or path).*

The point here is that we can develop any number of different characterizations of trees, yet they all represent the same abstract object. We would expect that a property which holds using one characterization must hold in an equivalent one. We can start to think about what types of properties hold for all of the above characterizations and what properties are dependent on the structure itself. It is this formalization of the structure of trees that we investigate here.

In order to formalize a structure, in this case the recursive types that represent trees, a uniform framework in which to reason is required. Here we use Nuprl [7], a mechanical theorem prover to assist us in our formalizations and our proofs. Nuprl is based on a constructive logic which allows for the specification and verification of properties, structures, and propositions. We prove that our data structures and the algorithms built on them are correct. Formalizing these data structures, functions, and properties in constructive type theory we can then use the proofs-as-programs feature of Nuprl’s type theory to extract correct-by-construction programs.

In section 2 we introduce the basic type theory needed to formulate simple trees. Using these primitive forms we will build the basic constructors and destructors necessary to formally reason about trees. We then, in sections 3 and 4, use these constructors to build a theory of trees and develop algorithms with these tree types in a formal setting. In section 5 we will discuss the notion of equality in constructive type theory and present theorems characterizing the decidability of equality. We base this on the decidability of the underlying type of the objects stored in the trees. Section 6 will further develop our tree structures in terms of what it means for trees to be equal.

## 2 Types in Constructive Type Theory

The notion of a type is fairly intuitive. Any computer scientist can easily identify a type. For instance,  $\mathbb{N}$ ,  $\mathbb{B}$ , or  $\mathbb{Z}$  are types, or in a programming language context, *bool*, *char*, *string*, *int*, *short*, *double*... are basic types. Additionally in a language such as ML, C or C++ we have the ability to build abstract data types using structs or building classes and objects. Type theory underlies the tools programmers use to build these abstract, or increasingly complex, types from more basic or atomic types. The formal development of complex types can be mapped directly onto similar notions in a programming language. A natural mapping is easier to construct when going from a type theory syntax into the syntax of a functional programming language. Yet programmers using an imperative language can benefit from understanding the construction of types as well and a similar intuition about types as formal structures and implemented abstract data types can be developed.[14]

First it would be handy to have a definition of type. Most definitions of types start with a set theoretical notion.[12] [15] We will never in the current treatment explicitly consider a type solely as a set. Yet, it is a nice construction to keep in mind as we develop certain forms of types.

**Definition :**

A completely defined type,  $T$ , is a set with operations which

- i*) determine for an element  $a$ ,  $a \in T$  and
- ii*) for elements  $a, b \in T$  make the judgement  $a = b \in T$ .

We say that a type is completely defined when it is the case that we know what the elements in the type look like and what it means for them to be equal. It could be the case that we may not be able to determine equality between two elements. In this case we say that the type is partially defined. Any type that has at least one element is said to be inhabited. Any type with no inhabitants is said to be void. We determine equality between two elements of a type if they have the same reduced form. In other words if we were to fully reduce (or evaluate) two terms to their simplest form we would see them to be identical. The structural form, the construction, and the extensional value would all be the same. We call this most reduced form the canonical form of the type.

### 2.1 Void and Unit

A natural starting point for a construction of more complex types is to introduce types that one would consider to be atomic.<sup>1</sup> We can consider these as types that cannot be constructed out of any simpler types. The empty type, **Void**, is defined as the type that has no elements in it, or no members. To say that something is in **Void** would be to assert an absurdity and thereby false. Although we will not need the empty type in our initial tree formalizations it is a basic type.

The next larger type is **Unit** which we designate as **1**. **1** is the type that contains one element. This single element is denoted as “ $\cdot$ ” and is referred to as “it”. Further we say

---

<sup>1</sup>Nuprl does not necessarily consider Unit.

$\cdot \in \mathbf{1}$  to indicate that  $\cdot$  is a member of the type  $\mathbf{1}$ . We can characterize the structure of any type by describing all of its elements. In the case of  $\mathbf{1}$  this is quite simple. We state:

$$\forall x. x \in \mathbf{1} \implies x = \cdot$$

The implication here is that if we find any element in  $\mathbf{1}$  that element must be identical to  $\cdot$ . Although it is an obvious statement it introduces the first example of a canonical form for a type. In this case we state that the canonical form of  $\mathbf{1}$  is the form of “it” or more exactly the element  $\cdot$ . In most cases the canonical form may be more complex but is the form which a fully evaluated element takes. (i.e. It’s extensional value.) For instance the canonical form of the natural numbers is simply the numbers themselves. (i.e. 0,1,2,... each being a canonical representation of the number they represent). [16]  $5 + 6$  is not in canonical form but 11, which is equivalent to the former, is.  $\mathbf{1}$  being a basic type, is not, by itself, particularly useful. However it can be combined with other connectives and types to build more complex types.

## 2.2 Disjoint Union

In order to build new types we introduce constructors which allow us to build these new types by combining previously defined types. The first constructor we introduce is the Disjoint Union of two types. Disjoint Union is denoted by  $+$ . We define the type formation rule as follows.

$$\frac{A : Type \quad B : Type}{A + B : Type}$$

Which reads as, if  $A$  is some type and  $B$  is some type, then  $A + B$  is also a type.  $A + B$  is well formed when  $A$  and  $B$  are well formed. Once again we can characterize the structure of the type by describing the form of the inhabitants of the type. The form of the elements in any type is dictated by their construction. In the case of a Disjoint Union of two types the injection functions will play the role as the constructors for the type. The injection functions are *inl* and *inr*, inject-left and inject-right respectively.

$$\frac{\Gamma \vdash A : Type, B : Type \quad \Gamma \vdash a \in A}{\text{inl}(a) \in A + B}$$

and

$$\frac{\Gamma \vdash A : Type, B : Type \quad \Gamma \vdash b \in B}{\text{inr}(b) \in A + B}$$

A good intuition about what these constructors mean is the following: if we are given an element  $a$  which is of type  $A$  we can construct an element of type  $A + B$  by labeling, or tagging,  $a$  as being in the left half of the disjunct. We have a similar notion for the right disjunct.

Computation with the Disjoint Union type consists first of making a decision as to whether a term  $t$  is of the form  $\text{inl}(t)$  or  $\text{inr}(t)$ . Then we substitute  $t$  into the body of the term we want to evaluate. Our operator for accomplishing this computation is called *decide* and takes the following form:

$$\text{decide}(t, x.t_1, y.t_2)$$

Here  $t$  is an element of type  $A + B$ ,  $t_1$  and  $t_2$  are arbitrary terms, and  $x$  and  $y$  are variables that may occur free in  $t_1$  and  $t_2$ , respectively. To evaluate this decide term we look at the form of  $t$ .

$$\text{decide}(\text{inl}(t), x.t_1, y.t_2) \rightarrow t_1[x := t]$$

The result of evaluating a term where  $t$  is of the form  $\text{inl}$  is the value of further evaluating  $t_1$  with all instances of  $x$  in  $t_1$  replaced with  $t$ . An  $\text{inr}$  term is similar:

$$\text{decide}(\text{inr}(t), x.t_1, y.t_2) \rightarrow t_2[y := t]$$

The terms  $x.t_1$  and  $y.t_2$  are binding structures for  $x$  and  $y$  in  $t_1$  and  $t_2$ , respectively. We would need to apply capture avoiding substitution to assure that no variables in  $t$  are duplicates of  $x$  or  $y$ .

Given only this we could begin to describe more complex and useful types. For instance we can easily define the booleans in terms of Unit and Disjoint Union.

$$\mathbb{B} = 1 + 1$$

where true is  $tt = \text{inl}(\cdot)$  and false is  $ff = \text{inr}(\cdot)$ . We can even begin to use the decide function as the basis for a small programming language. But we leave this to the motivated reader as an exercise.

## 2.3 Cartesian Product

The next type connective to be introduced is the Cartesian Product of two types which we denote as  $\times$ . Similar to the Disjoint Union a Cartesian Product is well formed when the two types that compose it are well formed. The introduction rule also takes a similar form, with  $A$  and  $B$  being arbitrary types.

$$\frac{A : Type \quad B : Type}{A \times B : Type}$$

Elements of type  $A \times B$  are ordered pairs of elements from  $A$  and  $B$ . The first element is from  $A$  and the second is from  $B$ . To construct an element of a Cartesian Product we apply the pairing function to an element from  $A$  and from  $B$ .

$$\frac{a : A \quad b : B}{\langle a, b \rangle : A \times B}$$

A Cartesian Product of types  $A$  and  $B$  consists of all the ordered pairs of elements from  $A$  and  $B$ .

Computation with elements of  $A \times B$  uses the projection functions ( $\pi_1$  and  $\pi_2$ ) to return either the first or second element of the pair. In order to get the first we apply the first projection to a pair:

$$\forall a : A, b : B. \pi_1 \langle a, b \rangle = a$$

and similarly the second projection:

$$\forall a : A, b : B. \pi_2 \langle a, b \rangle = b$$

We now have nearly everything we need to construct a type of trees. The **Unit** and **Void** types and the connectives given above can be used as a grammar for constructing new types. For instance  $1 + B \times B$ ,  $Void + Void$ ,  $A + B \times C + A \times B \times C$ , can easily be identified as types if it is known that  $A, B$ , and  $C$  are types. By combining these type connectives, the previously defined types, and the following introduction of recursive types we will be able to develop trees using these simpler constructions as building blocks.

## 2.4 Recursive Types

The usual method to define a recursive type is by the  $\mu$  least fixed-point operator.  $\mu$  takes a type variable,  $T$  for instance, and a type  $\phi$  where  $T$  may occur as a free variable in  $\phi$ . The syntax for the  $\mu$  operator is:

$\mu(T.\phi)$  where  $\phi$  is a well-formed type expression and  $T$  may occur as a free variable in  $\phi$

By this construction, occurrences of the variable  $T$  are bound in  $\phi$ . The least fixed point exists if  $T$  occurs only positively in  $\phi$ . [11] With the type constructors we have seen so far all occurrences are positive. (A common example of a negative occurrence would be  $A$  in the function type  $A \rightarrow B$ . But the function type is unnecessary for the current description. A development of it can be found elsewhere.[16]) Hence if the least fixed point describes a well formed type then any construction  $\phi$ , a type expression over  $+$ ,  $\times$ ,  $\mathbf{1}$ , and any arbitrary types  $A$ ,  $B$ , ... possibly with free occurrences of  $T$ , then  $\mu(T.\phi)$  is a well formed type.

Although recursive types could take any form consistent with the type construction described above, the polynomial form is the most common and we restrict our reasoning to types with this form. Since  $T^n$  can be written as  $T \times T \times T \dots \times T$  (n times) then a generic polynomial type,  $\phi$ , can be generically described by the following equation [9]:

$$\phi = 1 + T + T^2 + \dots + T^n$$

We say that  $\phi$  is of order n and  $\mathbf{1}$  is our **Unit** type. However it will be necessary to allow each term to be parameterized by a product of other type variables.

Let  $A_i = X_{i,1} \times X_{i,2} \times \dots \times X_{i,m}$   
 where each  $X_{i,j}$  is a type variable

A parameterized generic polynomial type is defined as

$$\phi = 1 + A_1 \times T + A_2 \times T^2 + \dots + A_n \times T^n$$

This form can be treated algebraically as would any other polynomial. But the type may not be a complete sequence from  $0 \dots n$ . Hence there further abstraction is necessary to allow for more freedom when describing our types.

Let  $l_i$  be a  $\mathbb{N}$ ,  
 then  $\phi(T) = 1 + A_1 \times T^{l_1} + A_2 \times T^{l_2} + \dots + A_n \times T^{l_n}$

$\phi(T)$  is a function which generates a polynomial representing our recursive type. We will generally leave the  $T$  in  $\phi(T)$  out and assume it is clear from the context which variable is bound in  $\phi$ .

Stating what it means to be a member of a recursive type in terms of a rule as we have for the other type constructions:

$$\frac{\Gamma \vdash T : Type \quad \Gamma \vdash t \in \phi[T := \mu(T.\phi)]}{\Gamma \vdash t \in \mu(T.\phi)}$$

which states that if  $T$  is a type and  $t$  is in an element of the type  $\phi[T := \mu(T.\phi)]$  (the polynomial described by  $\phi$  where every instance of  $T$  in  $\phi$  has been replaced by  $\mu(T.\phi)$ ) then it is also an element of  $\mu(T.\phi)$  and vice versa.

What is the relationship between  $\mu(T.\phi)$  and  $\phi[T := \mu(T.\phi)]$ ? The notion that two elements of a recursive type are equal can take on two different flavors. Equi-recursive equality asserts that these expressions, the type and the unfolded type, are “definitionally equal” or interchangeable. Since there is a conversion between a recursive type and its unfolding, the unfolded version can be used in place of the original and vice versa. Iso-recursive is slightly different. It considers a recursive type and the unfolded instances of it as not being directly interchangeable. However they are considered isomorphic to each other via the fold and unfold operators described above. Nuprl uses an equi-recursive approach and it is the responsibility of the user to prove well-formedness goals (type checking goals) to show that every unfolding is equivalent to the base recursive type. [13]

Functions can be applied to  $\phi$  which transform it in some way. For instance we can apply an unfold operator. Unfolding is the operation by which we replace the type  $T$  by the definition of the recursive type. In terms of the  $\mu$  operator:

$$\mu(T.\phi) = \phi[T := \mu(T.\phi)]$$

Or the right hand side is the result of replacing every free instance of  $T$  in  $\phi$  by  $\mu(T.\phi)$ . [6] The two sides are equal via the unfold operator (and it’s inverse the fold operator). We will also allow for a partial substitution within the structure allowing us to unfold certain portions of the structure while leaving the other sections intact. We will see examples of this shortly.

### 3 Trees

Any of a number of characterizations could have been chosen for trees. However the formulation of a recursive type gives us a structure that we can manipulate in a fashion synonymous with a programmatic manipulation. If we restrict our trees to the form of polynomials then the recursive types can be treated as algebraic structures. We can apply transformations to our trees such as fold and unfold. In the latter we get a deeper representation of the structure by going down one level in the recursive structure. In the former we simplify the structure by replacing a portion of the polynomial by a smaller but equivalent representation. [17] [6] We begin our investigation of trees by developing a recursive type representing unlabeled trees or *btrees*.

### 3.1 Unlabeled Trees

Using the recursive type constructor  $\mu, \mathbf{1}, \times, \text{and} +$  we can introduce a type of unlabeled trees. Unlabeled indicates that there is no information carried in the nodes of the trees. These trees are nodes with two subtrees or are empty trees, neither of which contain any explicit information. Contrary to an informal programming notion of trees, the empty tree is in fact a tree as we will see and hence contributes to the structure although it may not contribute anything in a resulting program derived from the recursive type. In other words it makes sense to have it in the algebra of trees as it is in fact the base case of our inductive scheme. Our unlabeled tree type is formed from the Disjoint Union of the unit type and the Cartesian Product of two trees, giving us our polynomial which embedded in our  $\mu$  operator gives us the recursive type. We denote it as:

$$\text{Btree} = \mu(B.1 + B \times B)$$

Informally we may write  $B = 1 + B \times B$ . As *btrees* are in fact a type there must be, as with all types, a way of constructing and destructing it. There are two constructors for *btrees*. The first constructs an empty *btree*. The second constructs a node *btree* when given two arguments which are also *btrees*. We formally state these as:

$$\begin{aligned} \text{Empty} &= \text{inl}(\cdot) \\ \text{Node}(l, r) &= \text{inr}(\langle l, r \rangle) \\ &\text{where } l \text{ and } r \text{ are btrees.} \end{aligned}$$

It is easy to see that  $\text{inl}(\cdot)$  and  $\text{inr}(\langle l, r \rangle)$  are in fact of type *btree*. Destructors are developed through case analysis on the form of *btree*, empty or node, where  $t$  is a *btree*.

case of( $t$ ) =

$$\begin{aligned} \text{Empty} &\rightarrow t_1 \\ \text{Node}(l, r) &\rightarrow t_2 \\ &\text{where } t_1 \text{ and } t_2 \text{ are terms and} \\ &\text{the computation rules are:} \end{aligned}$$

When  $t = \text{Empty}$

(case of( $\text{Empty}$ )) =

$$\begin{aligned} \text{Empty} &\rightarrow t_1 \\ \text{Node}(l, r) &\rightarrow t_2 \longrightarrow t_1 \end{aligned}$$

and when  $t = \text{Node}(m, n)$

(case of( $\text{node}(m, n)$ )) =

$$\begin{aligned} \text{Empty} &\rightarrow t_1 \\ \text{Node}(l, r) &\rightarrow t_2 \longrightarrow t_2[l := m, r := n] \end{aligned}$$

Given these constructors and destructors we can create predicates on trees. For instance, it is often the case that one would want to determine if a tree is empty or not. We can create

a predicate  $\text{Empty?}(t_1)$ , where  $t_1$  is an arbitrary tree, that will determine if  $t_1$  is of the form  $\text{inl}(\cdot)$  or not. We write this function as:

$$\text{Empty?}(t_1) = \text{decide}(t_1, tt, ff)$$

This syntax can get unruly when multiple decide statements are nested together. This is a problem remedied in the next section. There are several other predicates that take on a similar form to  $\text{Empty?}$ .  $\text{Leaf?}$ ,  $\text{Root?}$  and  $\text{Child?}$  are just a few examples, although they can get more complex.

### 3.2 Examples

The notion of a unlabeled binary tree should be fairly obvious. However a visual representation of the underlying structure can always be beneficial.

a)



The trees displayed in the examples are, in order, the empty tree ( $\text{Empty}$  or  $\text{inl}(\cdot)$ ); a tree with an arbitrary left and right subtree ( $\text{Node}(l, r)$  or  $\text{inr}(l, r)$ ); figure (1.c) is a branching tree with each nodes terminated by empty *btrees* and figure (1.d) is  $\text{Node}(l_1, \text{Node}(l_2, r_2))$  where  $l, r, l_1, l_2, r_2$  are all arbitrary btrees. We can now see how the constructors for *btrees* relate to the actual structure.

## 4 Formalization of btrees in Nuprl

There are many procedures and proof strategies that can be automated for trees and in fact for any recursive type. The pattern of development is almost always the same. Given a polynomial representing a recursive type, each term in the polynomial can represent one form the recursive structure can take. In our *btrees* we saw that given a type  $1 + \text{btree} \times \text{btree}$  the first term is an empty *btree* while the second was a *btree* with two subtrees which are also *btrees*. In the last section we built constructors for each of these polynomial terms. This can be done, in a similar fashion, for any parameterized generic polynomial type. It can in fact be automated in such a way that given any polynomial representing a recursive type, the constructors and methods of destructing a type can be created. This has been done in Nuprl. Given an abstract type (in this case the abstract type is in fact a recursive type, but could, in general, not contain any variables that are bound in its structure as in the  $\mu$  representation.) Representing recursive types in Nuprl is similar to the  $\mu$  construction. We use Nuprl's `simplerec` constructor which takes a variable,  $X$  and a type,  $\phi$ , that may have  $X$  occurring free in it. The Nuprl term has the structure

$$\text{simplerec}(X.\phi) \quad \text{where } X \text{ is a type variable and } \phi \text{ is a type expression}$$

Parsing the terms in  $\phi$ , constructors can be iteratively built for each term in the Disjoint Union of  $\phi$  ( corresponding to the terms in the polynomial). These can then be combined into a case statement that will allow unfolding and analysis of the structure by cases. It gives a concrete and readable form of the decide terms described above. In general a destructor is defined in terms of a case operation. For a type  $\phi = 1 + A_1 \times T^{l_1} + A_2 \times T^{l_2} + \dots + A_n \times T^{l_n}$  the corresponding destructor will look like:

$$\begin{aligned} \phi\text{-case}(\phi - term) = & \text{ case of}(\phi - term) \\ & \text{Constructor}_0 \rightarrow t_0 \\ & \text{Constructor}_1(args_1) \rightarrow t_1 \\ & \cdot \\ & \cdot \\ & \cdot \\ & \text{Constructor}_n(args_n) \rightarrow t_n \end{aligned}$$

Where each  $t_i$  is a term and  $args_i$  are  $A_{i1} \times \dots \times A_{im}$

The arity of the arguments,  $args$ , is dependent on the order of the term in the polynomial. The Nuprl abstract data type library contains meta-programs which generate these constructors and destructors automatically given the signature of the type. The Empty? predicate can be recast in terms of the case statement for *btrees*. For a *btree*  $t_1$  of the form  $\text{btree} = \mu(T.(1 + T \times T))$ , Empty? is defined as:

$$\begin{aligned} \text{Empty?}(t_1) = & \text{ btree-case of}(t_1) \\ & \text{Empty} \rightarrow tt \\ & \text{Node}(l, r) \rightarrow ff \end{aligned}$$

The case statement syntax is easier to comprehend than a complex series of nested decide statements. Btree-case is defined in Nuprl by the decide operator but when reasoning about *btrees* the case statement facilitates manipulating the various instances of *btrees*. From this point we will use the easier to read syntax which is definable in Nuprl using the display form mechanisms.[7]

Every generic recursive type is an inductive definition that has a corresponding scheme of structural induction. This structure can be used to define an induction principle based on the structure of the type. Much like the cases operation this induction principle is automatically generated by parsing the abstract type into its component terms. Nuprl has a defined induction principle for recursive types called `recElimination` which can be used here to generate both the base case(s) and an induction hypothesis based on a predicate about a smaller structure of the abstract type. For each recursive type a specific tactic can be built automatically which provides an inductive scheme for the type. This automation is invaluable since it gives a consistent method of doing proofs by structural induction on a specific recursive type. We call this, in the case of *btrees*, `btreeRecElim` or in a more general case  $\langle\langle TypeName \rangle\rangle\text{recElim}$ , giving both a consistent method and a tactic name for each recursive type.

## 4.1 More functions on btrees

Building on the predicates defined so far and using the ability to unfold a recursive definition by cases several algorithms and properties immediately come to mind that would be useful in characterizing any *btree*. Nuprl assists in the proof of these properties. Given that we are working with *btrees*, which have a rich literature supporting them, finding properties to formalize is hardly a chore. Selecting ones that can truly describe a structure and be used in multiple constructions is possibly a bit more challenging. A common tree property is its height. Using our *btree*-case operation height is defined in a natural way.

$$\begin{aligned} \forall t_1 : \textit{btree}. \\ \text{height}(t_1) = & \text{ case of}(t_1) \\ & \textit{Empty} \rightarrow 0 \\ & \textit{Node}(l, r) \rightarrow 1 + \max(\text{height}(l), \text{height}(r)) \end{aligned}$$

This states that if a *btree* is empty then it contributes nothing to the height of a *btree* but if it is of the form of a node with a left and right subtree the height from that point is 1 plus the maximum of the height of the left and the height of the right subtrees. This matches our intuition about the height of a *btree*.  $\text{Size}(t_1)$  can be defined in a similar fashion.

$$\begin{aligned} \forall t_1 : \textit{btree}. \\ \text{Size}(t_1) = & \text{ case of}(t_1) \\ & \textit{Empty} \rightarrow 1 \\ & \textit{Node}(l, r) \rightarrow 1 + (\text{Size}(l) + \text{Size}(r)) \end{aligned}$$

Here a slight deviation from our intuition may become evident which is actually inherent in the height case as well. In defining size, typically any node would be counted as contributing to the size of the tree, while empty trees might be ignored. However due to the fact that empty *btrees* are in fact of type *btrees* and part of the algebraic definition of what it means to be a *btree* they must be included. The same characteristic is hidden in our definition of height. A node with no proper subtrees (i.e. no non-empty subtrees) is typically considered a leaf. In the height function it should be noted that even if  $\max(\text{height}(l), \text{height}(r)) = 0$  (i.e. both  $l$  and  $r$  are empty) the existence of those empty trees which are necessary to terminate our recursive structure, do contribute to a level the height of the tree. In a typical programming implementation the empty trees would often be ignored but in a formal setting they are in fact an inherent part of the tree. There appears to be no good consensus on these definitions and how to handle the empty tree. In a language, such as ML, that supports tagged unions we will consider the empty tree as a part of the structure or the grammar that defines a tree. In a language such as C++ the notion is a little more cloudy. Given our example in the introduction, should we consider a null pointer as an empty tree or not? Examples abound on both sides in literature and the definitions of properties such as height and size vary accordingly. [8],[18]

In Nuprl any abstraction such as height or size would need to be shown to be well formed or in other words a member of a well formed type. Both of these properties, size and height, are in fact in  $\mathbb{N}$ . Proving this well-formedness goal requires reasoning about the structure of the *btree* and utilizes the `btrecElim` tactic. The base cases are automatically proven

in both cases and the inductive cases take 3 extra proof steps to verify that height and size always, for any arbitrary *btree*, return a natural number. However as the end goal here is to build a stronger definition of what properties characterize a *btree*, combining properties can be used to create this stricter notion of *btree*. Often further constraining these definitions gives us a secondary validation that our specifications of properties are in fact valid. Since size and height have been defined we can relate them to create a composite property which is well known.

$$\forall t : btree. \text{size}(t) \leq 2^{\text{height}(t)+1} - 1$$

We prove this by induction on  $t$ . Since this property is known to be true, by showing it in a formal setting gives further validation that we have correctly specified our trees. It should be noted that this holds for any definition of height and size. Hence we need to be careful to make sure that we are being consistent with respect to the base case (ie the empty *btree*) For instance if our size function included the existence of an empty *btree* but our height function did not consider it as contributing to the height of a tree, this theorem would hold. This is easily seen for the empty *btree* with  $\text{size}(\text{empty}) = 1$  or  $0$  and  $\text{height}(\text{empty}) = 0$

One final property of trees that can be presented is that of shape. This differs from the above properties by the fact that it takes into account two trees concurrently. Recursively, two *btrees* have the same shape if they are both empty or both have a root node and each of the subtrees have the same shape. We can formalize this into a proposition by using the *btree* case function. Arbitrarily the case analysis is done on the first *btree* of the pair.

$$\begin{aligned} \forall t_1, t_2 : btree. \\ \text{shape}(t_1, t_2) = & \text{case of}(t_1) \\ & \text{Empty} \rightarrow \text{Empty?}(t_2) \\ & \text{Node}(l, r) \rightarrow \text{shape}(l, \text{Left}(t_2)) \wedge \text{shape}(r, \text{Right}(t_2)) \end{aligned}$$

Where Left and Right return the left and right subtrees respectively. Since there is no additional information to compare in the *btree* if the proposition returns true there is an implication that the shape function defines equality on two *btrees*. In fact we can prove that this is in fact the true structural equality for the type of *btrees*. Stated formally

**Theorem 4.1**

$$\forall t_1, t_2 : btree. \text{shape}(t_1, t_2) \iff t_1 = t_2 \in btree$$

A proof and further discussion will be provided in a subsequent section.

## 4.2 Extending trees

We motivate our discussion about the equality of trees by introducing a new variety of tree. Extending the type of trees is a simple matter by the constructions we had above. The next logical step in tree development would be to create a tree that carries information in its nodes, or in other words a labeled tree. For an arbitrary type  $T$  the type of  $T$  trees will hold information of type  $T$  at each node. Hence the node constructor will now take a triple

instead of a pair,  $\langle x, l, r \rangle$  where  $x$  is of type  $T$ , and  $l$  and  $r$  are of type  $T$  tree giving the construction  $\text{Node}(x, l, r)$ . The type is formally defined as:

$$T \text{ tree} = \mu(S.1 + T \times S \times S)$$

In a fashion identical to *btrees* the constructors (with the addition noted above), the `treerecElim`, and the tree-case destructors are all automatically generated by Nuprl’s abstract data type mechanism. With this simple extension the majority of our propositions and functions that refer to structural properties of trees remain the same, only with the addition of an element of type  $T$  to each node. The most glaring exception to this is the notion of equality between two  $T$  trees. In this case we will need to consider elements of  $T$  along with the shape of the tree.

## 5 Constructive Equality

It is not our intention to give an exhaustive description of equality in a constructive type theory. Those interested would be advised to look in [10],[16], [15], [3] [12], and [7] with the second being the most basic. Equality in Constructive Type Theory (CTT) is fundamentally different than in a classical or set theoretical setting. In set theory, two sets are equal if and only if for every member of one set there is an equivalent member in the other set. Underlying this is the implicit assumption we can determine that  $\forall x, y : T. x = y \vee (x \neq y)$ . (ie we assume that two elements from a set(s) are equal or they are not.) CTT requires this method of deciding equality to be explicit. One element of a type is equal to another element of the same type if they have identical constructions, or can be reduced to the same canonical form. We mentioned earlier the notion of a canonical element without giving a formal definition of what that entails. We can consider a canonical element as one that is the irreducible value of some program. For instance *true*, 5,  $\lambda x.x$  are all in their respective canonical forms for their type. By our example above  $5 + 6$  can be reduce to 11, and “if  $3 = 9$  then true else false” can be reduced as well and hence are not in canonical form. [12]

If we were to consider the canonical forms for the elements in the Booleans they would be the values `tt` and `ff`. We also have a method of constructing these elements. Namely  $tt = \text{inl}(\cdot)$  and  $ff = \text{inr}(\cdot)$ . A list of natural numbers is constructed by using the `cons` operator and the empty list to build ever larger lists.

$$\text{cons}(5, (\text{cons}(8, (\text{cons}(4, [])))))) = [5; 8; 4]$$

The  $\text{cons}(x : \mathbb{N}, \text{Nlist})$  operator and the empty list `[]` are the constructors for lists just as *Empty* and *Node*( $l, r$ ) are constructors for *btrees*. We could abstract this even further by constructing the natural numbers out of 0 and *succ*, the successor function. Any element we build out of these constructors will be unique and hence be in the canonical form for that type.

How does this lend itself to equality in CTT? The basis of Martin-Lofs type theory (which is constructive) revolves around making judgments about types and elements of types. Here we are concerned with one judgment in particular, namely when are two elements of a particular type equal. Or more exactly, what does it mean to say that two elements  $a : A$  and

$b : A$  are equal in type  $A$ ? We denote this as  $a = b \in A$  or  $a =_A b$ . The logic behind CTT contains sets of rules that guide us in generating proofs that certain propositions hold. Or in other words constructing a witness to the validity of that proposition. One type of rule that is often used is the “Introduction Rule”. Every type has an introduction rule or rules and they are exactly our constructors for a type with perhaps a slight variation in notation. For example the list type (parameterized by an arbitrary type  $A$ ) has two introduction rules. They are synonymous with the constructors described above. From these rules any list over the type  $A$  can be constructed. In particular every construction is unique in that no  $\text{List}(A)$  can be constructed from more than one sequence of the introduction rules and parameters of type  $A$ .<sup>2</sup>

**Definition** Introduction Rules for  $\text{List}(A)$

$$\frac{}{[] \in \text{List}(A)} \quad []\text{-introduction}$$

$$\frac{\begin{array}{l} A : \text{type} \\ a \in A \\ l \in \text{List}(A) \end{array}}{\text{cons}(a, l) \in \text{List}(A)} \quad \text{cons - introduction}$$

We can develop a similar set of formal rules for *btrees* and *T trees*. They are once again exactly our constructors for the type.

**Definition** Introduction Rules for *btree*

$$\frac{}{\text{Empty} \in \text{btree}} \quad \text{Empty - introduction}$$

$$\frac{l \in \text{btree} \quad r \in \text{btree}}{\text{Node}(l, r) \in \text{btree}} \quad \text{Node - introduction}$$

*T tree* is similar except we need to assert that  $T$  is a type and  $x$  is of type  $T$  in the *Node - introduction* case.

In recursive types, such as lists and trees, deciding equality takes an extra effort. As was mentioned earlier, recursive equality is approached in two fashions, Iso-recursive and equi-recursive equality. It was also noted that Nuprls approach is equi-recursive. In the equi-recursive approach equality is derived from the type checker verifying that an unfolded structure is of the same type as the original. If a structure is derivable from a finite number of unfoldings (substitutions and unfoldings of the form  $\mu(T.\phi) = \phi[T := \mu(T.\phi)]$ ) then the two structures are convertible and therefore equivalent. Then equality for the recursive type is derived in part from the equality notion given by the other type constructors existing in the polynomial  $\phi$ : Disjoint Unions, Unit, and Cartesian Products. However as we have seen in *T trees* a recursive type can be parameterized by an arbitrary type  $T$ . Any type, recursive or not, parameterized by, or constructed from, another type must consider what it means to

---

<sup>2</sup>We make a presumption of dealing only with finite structures at this point as Nuprl is limited to these, or at least finite lists

be equal in the type  $T$  along with the notion of equality in its own structure. We will see that it is often the case that what we can say about  $T$  equality determines what we can say about  $T$  tree equality.

Now it is possible to say exactly what is meant by  $a = b \in B$ , given an arbitrary type  $B$ , and two elements  $a$  and  $b$ , both of which are well formed in  $B$ . If both  $a$  and  $b$  are reduced to their canonical form then  $a = b \in B$  if they have the same constructors or same construction in the cases where multiple applications of the introduction rules have taken place. [3] In a recursive type we have further determined that the structures of  $a$  and  $b$  are equi-recursively equal (or iso-recursively equal). Or from a computational point,  $a$  and  $b$  reduce to the same canonical value. [12] But this raises the question of how do we know that they are in canonical form and further that the constructions are the same. Or if we would rather work on a higher algorithmic level than reducing everything down to the constructors of the type (i.e. a sequence of *inl*, *inr*,  $\pi_1$ , and  $\pi_2$  terms) what can we then say about equality?

Much of this can be taken care of by developing a decision procedure, or algorithm, to decide if elements are equal in a type. If equality is decidable (i.e there is a decision procedure for equality in the type) then we say the type is *Discrete*. As an example of a type that is not discrete, consider the functions  $\mathbb{N} \rightarrow \mathbb{N}$  which is known to be undecidable and hence not discrete. In order for an algorithm to be considered an decision procedure which determines equality between two elements of a type it must do two things. First it must be shown that the decision procedure models the equality defined by the type in terms of the canonical elements. In other words, a decision procedure  $Eq_A$  for a type  $A$  is sound with respect to the type such that for every two elements  $a$  and  $b$  of type  $A$ ,  $Eq_A$  will return true (or “yes”) on  $a$  and  $b$  if and only if  $a = b \in A$ . Formally we would need to prove

$$\forall a, b : A. Eq_A(a, b) \iff a = b \in A$$

Or in terms of being discrete, the decision

$$\forall a, b : A. (a = b \in A \vee \neg(a = b \in A))$$

must be computable. In a constructive context this means that we must be able to find a positive answer to one of the disjuncts and more importantly be able to tell which one it is and how we know that it is true. The “how” is the second responsibility of a decision procedure.  $Eq_A$  must provide a witness, proof, to why the two elements are equal if in fact they are.

## 6 Tree equality

Equality on trees can now be exactly defined. Two *btrees* are equal if their constructions are the same, such that for every Node construction in one *btree* there exists a Node in the second *btree* which occurs at the same point in the sequence of the construction. Hence there exists a one-to-one correspondance when we traverse the *btrees* in parallel. Similarly the Empty constructor in one *btree* must have an equivalent in the second *btree*. This is the true structural equality when we make the judgement about two *btrees*; for  $t_1, t_2 : btree.t_1 = t_2 \in btree$ . Rather than reducing everything to the bare constructors, a higher level decision

procedure can be developed. In the case of *btrees* this has already been done in the terms of the *shape* proposition described earlier. We can prove theorem 4.1. First though we give the definition again:  $\text{shape}(t_1, t_2)$

$$\begin{aligned} \text{shape}(t_1, t_2) = \text{case of } t_1 \\ \text{Empty} \rightarrow \text{Empty? } t_2 \\ \text{Node}(l, r) \rightarrow \text{shape}(l, \text{Left}(t_2)) \wedge \\ \text{shape}(r, \text{Right}(t_2)) \end{aligned}$$

And restate the theorem.

**Theorem 4.1.** Shape is btree equality

$$\forall t_1, t_2 : \text{btree}. \text{shape}(t_1, t_2) \iff t_1 = t_2 \in \text{btree}$$

**Proof** The proof consists of showing that for every possible structure on *btrees* the algorithmic equality will hold if in fact they are equal in the type. It is done by induction on the structure of *btrees*.

$\Leftarrow$  In the base case, if both  $t_1$  and  $t_2$  are empty *btrees*, they are equivalent by construction. If  $t_1$  is empty but  $t_2$  is not of the form of an empty *btree*, `Empty?` returns false. The induction case is easy as well since we have assumed the built-in equality on *btrees* we can substitute  $t_1$  in for  $t_2$ .

$$\Gamma, t_1 = t_2 \in \text{btree}, t_1 = \text{Node}(l_1, r_1) \in \text{btree}, \text{Node}(l_1, r_1) = t_2 \in \text{btree} \vdash \text{shape}(\text{Node}(l_1, r_1), t_2)$$

Substituting into the conclusion and subsequent hypotheses, and thinning we get:

$$\Gamma, t_1 = \text{Node}(l_1, r_1) \in \text{btree}, \text{Node}(l_1, r_1) = t_2 \in \text{btree} \vdash \text{shape}(\text{Node}(l_1, r_1), \text{Node}(l_1, r_1))$$

Where  $\Gamma$  is a list of other hypotheses. In Nuprl some of the hypotheses would be thinned out but are left in here for clarity. Unfolding `shape` we are left to show two subgoals, namely  $l_1$  and  $r_1$  have the same shape. Using our induction hypothesis and substituting in with  $l_1$  and  $r_1$  it is easily proven in each subgoal.

$\Rightarrow$  **Base Case** Let  $t_1$  be an empty *btree*. ( $\text{shape}(\text{Empty}, t_2)$ ) Decomposing  $t_2$  we get two cases. Doing the substitution of  $t_2$

$$\Gamma, t_2 = \text{empty} \in \text{btree}, \text{shape}(\text{empty}, \text{empty}) \vdash \text{empty} = \text{empty} \in \text{btree}$$

Which is true. Otherwise

$$\Gamma, t_2 = \text{Node}(l, r) \in \text{btree}, \text{shape}(\text{empty}, \text{Node}(l, r)) \vdash \text{empty} = \text{Node}(l, r) \in \text{btree}$$

which upon unfolding and reducing `shape` on the left hand side derives false, hence the sequent is true.

**Induction Case** Using our `btreerecElim` tactic we can generate an induction hypothesis which allows us to assume that for any *btree*,  $t'$ , structurally smaller than  $t$  we can assume

that a property  $P$  holds. So in this case we get an induction hypothesis of the following form for a proposition  $P$ :

$$\forall t'_1, t'_2 : btree, P[t'_1] \wedge P[t'_2] \implies P[\text{Node}(t'_1, t'_2)]$$

In our case it takes the form of a function with a predicate on *btrees*.

$$\forall t'_1 : \{v : btree \mid uv\} \rightarrow (\forall t'_2 : btree. \text{shape}(t'_1, t'_2) \implies t'_1 = t'_2 \in btree)$$

where  $u$  is a predicate on *btrees* that shows  $v$  to be a *btree* with a certain property. Here the property asserts that the components of our decomposed tree ( $t_1$ ) are structurally smaller. We are allowed to assume that the induction hypothesis is true for them and any other arbitrary tree.

Using this induction hypothesis and the fact that our original *btree*  $t_1$  was decomposed into two subtrees of  $t_1$ , hence  $t_1 = \text{Node}(lt'_1, rt'_1)$  where,  $lt'_1 : \{v : btree \mid uv\}$  and  $rt'_1 : \{v : btree \mid uv\}$ , we can then proceed with finding the appropriate  $t'_2$  *btree* components. This is easily done by simply decomposing  $t_2$  into its component parts. We get two cases once again:

$$\Gamma, t_2 = \text{empty}, \text{shape}(\text{Node}(lt'_1, rt'_1), \text{empty}) \vdash \text{Node}(lt'_1, rt'_1) = \text{empty} \in btree$$

and

$$\begin{aligned} \Gamma, t_2 = \text{Node}(lt'_2, rt'_2), \text{shape}(\text{Node}(lt'_1, rt'_1), \text{Node}(lt'_2, rt'_2)) \vdash \\ \text{Node}(lt'_1, rt'_1) = \text{Node}(lt'_2, rt'_2) \in btree \end{aligned}$$

The case where  $t_2$  is empty quickly derives a contradiction, as in the base case. Taking  $t_2 = \text{node}(lt'_2, rt'_2)$  we decompose our induction hypothesis twice. First we do it with  $lt'_1$  and  $lt'_2$  to create one equality  $lt'_1 = lt'_2 \in btree$  and a similar decomposition for the right hand side. Our conclusion or proof goal states

$$\text{Node}(lt'_1, rt'_1) = \text{Node}(lt'_2, rt'_2) \in btree$$

We simply substitute the type equalities  $lt'_1 = lt'_2 \in btree$  and  $rt'_1 = rt'_2 \in btree$  into our conclusion which gives us

$$\text{Node}(lt'_1, rt'_1) = \text{Node}(lt'_1, rt'_1) \in btree$$

proving the theorem.  $\square$

For *btrees* the *shape* function is equivalent to the built-in equality for the type, or the type equality. The `recElim` tactics are extremely useful in proving such goals. In later proofs we will not go into as much detail on the statement of the induction hypothesis but note that it often follows the same pattern as seen above.

## 6.1 T tree equality

This raises the question about deciding equality for  $T$  trees. Can we define equality in an equivalent fashion to  $btrees$ ? Obviously there is extra information that we need to account for in a  $T$  tree that does not exist in the  $btree$ . This type  $T$  must have been well formed and will have its own built-in equality in the type theory. Taking this into account we can define an equality on  $Ttrees$  in a similar fashion to the shape “equality” on  $btrees$ . Consider the following function,  $treeEQ$ , defined by recursion on the structure of its first argument.

$$\begin{aligned} treeEQ(t_1, t_2) = & \text{ case of}(t_1) \\ & Empty \rightarrow Empty?(t_2) \\ & Node(x, l, r) \rightarrow (x = Node\_val(t_2) \in T) \wedge \\ & \quad (treeEQ(l, Left(t_2))) \wedge \\ & \quad (treeEQ(r, Right(t_2))) \end{aligned}$$

Where  $Node\_val$  is a function which returns the first element of the triple making up the  $T$  tree node.  $Node\_val$  will return an element of type  $T$ . The question must now be asked whether or not this is a decision procedure for the type  $T$  tree that is equivalent to the built-in equality given with the type of  $T$  tree. Intuitively it is, just as shape determined equality in  $btrees$ . The root node of both trees are checked, if the  $Node\_val$  of the two is the same then we check the left and right subtrees for equality using the same process. We can determine easily that this is in fact a well formed proposition by induction on  $T$  trees. However the first step in verifying equality is a check for equality in type  $T$ . It is the case that if  $T$  is not a discrete type then  $T$  tree can not be decided using this decision procedure. Without a discrete equality on  $T$  it would be impossible to show that  $treeEQ$  does in fact model the equality on the type. Yet we can show the following theorem.

### Theorem 6.1 (treeEQ is T tree equality)

$$\forall T : type. \forall t_1, t_2 : T \text{ tree. } treeEQ(t_1, t_2) \iff t_1 = t_2 \in T \text{ tree}$$

**Proof Sketch** This proof was done in Nuprl.

The proof of the  $\Leftarrow$  direction is quite easy. Since we have the structural equality  $t_1 = t_2 \in T$  tree we can substitute  $t_1$  for  $t_2$  and prove  $treeEQ(t_1, t_2)$ . We rely on the fact that tree equality defines an equivalence class, and in particular that it is reflexive. The statement of that theorem is as follows.

### Theorem 6.2 (TreeEQ is equivalence class)

$$\forall T : type. \forall t_1, t_2, t_3 : T \text{ tree.}$$

$$\begin{aligned} & treeEQ(t_1, t_1) \wedge \\ & (treeEQ(t_1, t_2) \implies treeEQ(t_2, t_1)) \wedge \\ & (treeEQ(t_1, t_2) \wedge treeEQ(t_2, t_3) \implies (treeEQ(t_1, t_3))) \end{aligned}$$

The proof of theorem 6.2 was only difficult in the transitive case and was done by induction on  $t_1$ .

$\implies$  This direction was much more difficult. Assuming  $treeEQ(t_1, t_2)$  gave us very little direct information about how  $treeEQ$  related to equality in  $T$  trees. Several lemmas were required that allowed for construction of the appropriate tree elements. Namely

**Lemma 6.2.1 (Empty tree Equality)**

$$\forall T : \text{type}. \forall t_1 : T \text{ tree}. \text{treeEQ}(t_1, \text{empty}) \implies t_1 = \text{Empty} \in T \text{ tree}$$

and

**Lemma 6.2.2 (Node tree Equality)**

$$\begin{aligned} \forall T : \text{type}. \forall x : T, \forall t_1, l, r : T \text{ tree}. \\ \text{treeEQ}(t_1, \text{Node}(x, l, r)) \implies \\ \exists y : T. \exists l1, r1 : T \text{ tree}. x = y \in T \wedge \\ l = l1 \in T \text{ tree} \wedge \\ r = r1 \in T \text{ tree} \wedge \\ \text{treeEQ}(t_1, \text{Node}(y, l1, r1)) \end{aligned}$$

With these two lemmas in conjunction with induction on the structure of the  $T$  trees the  $\text{treeEQ}$  is shown to be equivalent to the type equality. The proof in Nuprl is approximately 20 steps in length with the support of the Lemmas 6.2.1 and 6.2.2.  $\square$

This theorem creates a profound difference in the ability to prove other properties. Since we now have that our tree equality is in fact the equality defined by the construction of elements of the type, rewriting and substitution are possible. We can now state more clearly whether or not our  $T$  trees have a decidable equality.

**6.2 T Trees are discrete if T is discrete**

As one may expect, the decision procedure from  $T$  trees is dependent on what one can state about equality over the type  $T$ . In the lemmas 6.2.1 and 6.2.2 it was not implicit that  $T$  was discrete. All that was known is that there is some equality over  $a, b : T$  such that  $a =_T b$  can be stated as a proposition and that it is sensible to do as such.

If the assumption  $\text{Discrete}(T)$  is made then it is easy to see that  $\text{Discrete}(T \text{ tree})$ . The statement of the  $\text{treeEQ}$  algorithm covers two tasks. First it is checking the equality of the corresponding node values over type  $T$ . Second it is checking the shape to make sure that there is a one to one correspondence of the nodes. This second part is decidable on a finite  $T$  tree. However the first part implies that  $T$  tree equality is decidable if  $T$  is discrete. The theorem concerning decidability of  $T$  trees requires that we explicitly state  $\text{Discrete}(T)$ . We denote a discrete equality,  $eq$ , over type  $T$  in Nuprl as  $eq : \{= T_2\}$ .

**Theorem 6.3 (Discrete(T Tree))**  $\forall T : \text{type}. \forall eq : \{= T_2\}. \forall t_1, t_2 : T \text{ tree}. \text{Discrete}(\text{treeEQ}(t_1, t_2))$

The proof consists of doing induction over  $t_1$  and showing that we can always decide if  $(t_1 =_{T \text{ tree}} t_2) \vee \neg(t_1 =_{T \text{ tree}} t_2)$ . The proof often relies on analysis of the possible cases.

**6.3 T not known to be discrete**

What if it is not known whether  $T$  is discrete or not? If we make the assumption that  $T$  tree is in fact discrete without any consideration of  $T$  we have an interesting result regarding the discreteness of  $T$ . Although this is a rather odd assumption to make, given that we just spent time trying to explicitly state that  $T$  trees decidability is dependent on that of  $T$ . Here

we show that the discreteness of  $T$  and the discreteness of  $T$  tree are equivalent properties. We state and provide a sketch of the proof of:

**Theorem 6.4 ( Discrete (T) iff Discrete(T tree))**

$\forall T : type.(Discrete(T) \iff Discrete(Ttree))$

**Proof sketch**  $\implies$  In this case the proof is easy. Assuming that T is Discrete, we simply use the decision procedure given in *TreeEQ* to decide T tree.

$\impliedby$  Now if we assume Discrete (T tree) we know that we have a decision procedure for T trees. It may be different from the TreeEQ algorithm. Call this decision procedure A. We must construct a decision procedure for T. We do this by a reduction from T to T tree. The procedure is as follows.

Given  $a, b : T$

Construct  $t_1 = (Node(a, Empty, Empty))$  and  $t_2 = (Node(b, Empty, Empty))$

Run A on  $t_1$  and  $t_2$

The answer from A is the answer to the proposition  $a = b \in T$ .

By constructing the two trees from the elements in  $T$  we solve the problem of deciding  $T$  by deciding T tree.  $\square$

## 7 Future Work

As mentioned in the last section, constructive notions of equality and more to the point membership have a slightly different character than a standard classical definition. We are planning to investigate what implications these witnesses will have on the issue of membership and how can we characterize this in across a broad spectrum of recursive types.

The second direction that will be investigated is a deeper and more concrete look at how we are formalizing data structures. And further how can this assist us in creating correct by construction code based on our proofs and related specifications of the data structures. Using trees as a basis, we can start to look at more specific implementations of trees which have a specific purpose but for which much of what we have already shown will still hold. Red-black trees and binary search trees are two such specific data structures. There are also innumerable algorithms that can be formulated in Nuprl or another theorem prover.

## References

- [1] Helm project: Hypertextual electronic library of mathematics. <http://helm.cs.unibo.it/>.
- [2] Nuprl, prl automated reasoning project. <http://www.nuprl.org>.
- [3] Roland C. Backhouse, Paul Chisholm, Grant Malcolm, and Erik Saaman. Do-it-yourself type theory. *Formal Aspects of Computing*, 1:19–84, 1989.
- [4] Claude Berge. *The Theory of Graphs*. Wiley, New York, 1962.

- [5] Mark Bickford. Binary trees, nuprl libraries.  
[http://www.cs.cornell.edu/Info/People/sfa/Nuprl/mb\\_tree/Welcome.html](http://www.cs.cornell.edu/Info/People/sfa/Nuprl/mb_tree/Welcome.html).
- [6] F. Cardone and M. Coppo. *Decidability properties of recursive types. Lecture Notes in Computer Science, Volume 2841*. Jan 2003.
- [7] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock and N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986. Up to date information available at <http://www.nuprl.org>.
- [8] Guy Cousineau and Michel Mauny. *A Functional Approach to Programming*. Cambridge University Press, Cambridge, 1998.
- [9] Marcelo Fiore. Isomorphisms of generic recursive polynomial types. *Proceedings of the ACM POPL conference*, pages 77 – 88, 2004.
- [10] Per Martin-Lof. *Intuitionistic Type Theory*. Bibliopolis, Naples, Italy, 1984.
- [11] Yiannis N. Moschovakis. *Elementary induction on abstract structures*. North Holland, Amsterdam, 1974.
- [12] Bengt Nordstrom, Kent Petersson, and Jan M. Smith. *Programming in Martin-Lof's Type Theory. An Introduction*. Clarendon Press, Oxford, 1990.
- [13] Benjamin Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, 2002.
- [14] I.H. Poernomo, J.N. Crossley, and M. Wirsing. *Adapting Proofs-as-Programs*. Springer-Verlag, New York, 2005.
- [15] Giovanni Sommaruga. *History and Philosophy of Constructive Type Theory*. Kluwer Academic, Dordrecht, 2000.
- [16] Simon Thompson. *Type Theory and Functional Programming*. Addison-Wesley, Wokingham, England, 1991.
- [17] Tjark Weber and James Caldwell. Constructively characterizing fold and unfold. 2003.
- [18] Mark A. Weiss. *Data Structures and Problem Solving Using C++*. Addison-Wesley, Reading, MA, 2000.