

Constructive Membership and Indexes in Trees

Josef Pohl

Department of Computer Science
University of Wyoming

January 14, 2009

Abstract

Trees carrying information stored in their nodes are a fundamental abstract data type. Approaching trees in a formal constructive environment allows us to realize properties of trees, inherent in their structure. Specifically we will look at the evidence provided by the predicates which operate on these trees. This evidence, expressed in terms of logical and programming languages, is realizable only in a constructive context. In the constructive setting, membership predicates over recursive types are inhabited by terms indexing the elements that satisfy the criteria for membership. In this paper, we motivate and explore this idea in the concrete setting of lists and trees. We first provide a background in constructive type theory and show relevant properties of trees. We present and define the concept of inhabitants of a generic shape type that corresponds naturally and exactly to the inhabitants of a membership predicate. In this context, $(\lambda x. True) \in S$ is the set of all indexes into S , but we show that not all subsets of indexes are expressible by strictly local predicates. Accordingly, we extend our membership predicates to predicates that compute and hold the state “from above” as well as allow “looking below”. The modified predicates of this form are complete in the sense that they can express every subset of indexes in S . These ideas are motivated by experience programming in Nuprl’s constructive type theory and the theorems for lists and trees have been formalized and mechanically checked.

1 Introduction

Membership predicates and indexes into recursive structures are not new concepts. The connection between the two notions has never been approached constructively nor formalized until [11]. We make this connection in order to further show that indeed type theory can be used to provide a methodology that combines the functional capabilities of a strongly typed programming language and the strength of a theorem proving environment. There has been a number of efforts in recent years to bring programming languages up to the level

of a formal constructive type theory environment, namely the languages Epigram [25] and Omega [32, 33]. These languages are using the power of the Curry-Howard Isomorphism to relate their dependently typed programming environments to that of the logical foundations of constructive type theory. In this paper we describe a move in the opposite direction, from constructive type theory into the realm of practical programming.

In this paper, we work in Nuprl, [13] a Martin L of dependent type theory with recursive types and extensional function types. The best and most current account of Nuprl’s type theory can be found online [3]. Nuprl, as well as many other theorem provers such as Coq [6], Lego [21] and Alf[27] have a long history of using and applying the Curry-Howard isomorphism to develop verified software. We apply it here to the problem of programming with proofs. Examples of this application of Nuprl can be found in [31, 17, 7, 8, 10, 9, 20].

1.1 Inhabitants of Membership Predicates are Indexes

In constructive type theory, proofs implicitly contain programs. In this paper, we apply the propositions-as-types and proofs-as-programs interpretations to examine the structures of the inhabitants of membership predicates over recursively defined types. Under examination, we realize natural generalizations that make these predicates more expressive in terms of the collections of inhabitants they may contain.

Classically, given a structure S of type \mathcal{S} and some element x , a membership predicate $x \in S$ may be true or false. In the constructive setting, if the predicate is true, it is inhabited by the indexes into S leading to the element x . For example, if the structure is of type $\mathbb{Z} \text{ List}$ and S is the list $[1; 2; 2; 3; 2]$ then, not only is $2 \in S$ true, but its truth is witnessed by the indexes (whose form depends on how the predicate itself is specified) to the second, third and fifth elements of S . When the proposition $2 \in S$ is considered through the propositions-as-types interpretation, it is a type whose elements are the indexes of 2 in S . For tree-like structures, these indexes essentially correspond to paths in the tree. This interpretation of membership predicates as index types arises completely naturally in the constructive setting¹ in the following sense, we prove that the identity function is an isomorphism between a membership predicate and a shape type.

List and tree examples serve to hone intuition and provide a framework for considering some interesting questions related to these ideas in a well understood setting.

In general, we are interested in how the Curry-Howard isomorphism (*i.e.* the propositions-as-types and proofs-as-programs interpretations) can be exploited to explore design spaces. From a methodological view, we show how the identification of membership predicates with index types guides the development of the ideas presented here.

¹Indeed we can argue that the only way a membership predicate fails to be an index is if the index information is explicitly discarded in the specification of the membership predicate itself.

1.2 Related Work

Going back to Jay [19, 18], ideas about shape and polymorphism with respect to it have suggested that the separation of shape from content for recursive types may support generic programming. In that work, Jay proposed viewing recursive types as pairs consisting of a type of indexes together with a list of data elements. Membership functions are not strictly shape polymorphic [26], since indexes to the members depend, not only on the shape of the structure, but on the contents as well.

There has been a recent flurry of interesting work driven from categorical semantics for type theory [1, 4, 2]. These authors are investigating what they call *indexed containers* and what have been called *dependent polynomials* in [15] and polynomial recursive type [14]. Further we find that McBrides work in type derivatives [24] has a direct correlation to our notion of indexes. We believe that the work described here can be lifted to the more general setting and intend to do so in future work.

The next section describes the foundations of constructive type theory as used in this paper. This is followed up in section 3 by an exposition of constructive equality. In section 4 we describe the membership predicates, indexes, and shape types and how they relate to each other. Section 5 describes the methods by which we make the predicates fully expressive. We delve into a few notable programming considerations in section 6. Throughout the paper we use and apply constructive type theory to examine the constructive content of membership predicates as indexes.

2 Type Theory, Recursive Data Types, and Trees

The notion of a type is fairly intuitive. For instance, \mathbb{N} , \mathbb{B} , or \mathbb{Z} (the natural numbers, booleans, and integers, respectively) 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.[30]

Definitions of types derive from a set theoretical notion as described in [28] [34]

Definition 1 (Type)

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.

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.² 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, we say it is uninhabited. 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 $\cdot \in \mathbf{1}$ to indicate that \cdot is a member of the type **1**. We can characterize the structure of any type by describing all of its elements. In the case of **1** this is quite simple. We state:

$$\forall x : \mathbf{1}. x = \cdot$$

The implication here is that if we find any element in **1** that element must be identical to \cdot . **1** and **0** can be combined with other connectives and types to build more complex types. These other connectives form types, however per the Curry-Howard Isomorphism they also have a corresponding logical interpretation.

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.

$$\frac{A : \mathbb{U} \quad B : \mathbb{U}}{A + B : \mathbb{U}}$$

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. \mathbb{U} designates the universe of types at level i . In Nuprl, a judgement of the form above must generally be shown to be well formed. A well formedness goal can simply be seen as a requirement to prove that the proposed type is in

²Nuprl does not necessarily consider Unit as atomic.

fact a member of some universe level in the hierarchy of types. For the above instance we could prove the fact:

$$\forall A, B : \mathbb{U}. A + B \in \mathbb{U}$$

All well formedness goals, that we will encounter, will take this form. 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.

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

and

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

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 *inl* is the value of further evaluating t_1 with all instances of x in t_1 replaced with t . An *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, \mathbb{B} , 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.

$$\frac{A : \mathbb{U} \quad B : \mathbb{U}}{A \times B : \mathbb{U}}$$

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 (π_1 and π_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 μ least fixed-point operator. μ takes a type variable, T for instance, and a type ϕ where T may occur as a free variable in ϕ . The syntax for the μ operator is:

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

By this construction, occurrences of the variable T are bound in ϕ . The least fixed point exists if T occurs only positively in ϕ . 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 $A \rightarrow B$. [22] But the function type is unnecessary for the current description. A

development of it can be found elsewhere.[35]) Hence if the least fixed point describes a well formed type then any construction ϕ , 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, ϕ , can be generically described by the following equation [14]:

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

We say that ϕ 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.

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

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.

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

$\phi(T)$ 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 ϕ .

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:

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

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 ϕ where every instance of T in ϕ 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 below. Nuprl uses a 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 [29].

Functions can be applied to ϕ 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 μ 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 ϕ by $\mu T.\phi$ [12]. The two sides are equal via the unfold operator (and it's inverse the fold operator). We will also allow for a partial substitution with in the structure allowing us to unfold certain portions of the structure while leaving the other sections intact.

2.5 Binary 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. This process of folding and unfolding is similarly described in [36] [12]. We begin our investigation of trees by developing a recursive type representing labeled trees.

Using the recursive type constructor μ , $\mathbf{1}$, \times , and $+$ we can introduce a type of binary trees. We allow information to be stored at each node. These trees are nodes with two subtrees or are empty trees, where the node will contain an an element of some arbitrary type. 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 tree type is formed from the Disjoint Union of the unit type and the Cartesian Product of an element of type A and two trees, giving us our polynomial which embedded in our μ operator gives us the recursive type. We denote it as:

$$tree_A = \mu S.(1 + A \times S \times S)$$

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

$$\begin{aligned} \text{Empty} &= \text{inl}(\cdot) \\ \text{Node}(a, l, r) &= \text{inr}(\langle a, \langle l, r \rangle \rangle) \\ &\text{where } l \text{ and } r \text{ are } trees \text{ and } a \text{ is of type } A \end{aligned}$$

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

case of(t) =

$$\begin{array}{ll} \text{Empty} & \rightarrow t_1 \\ \text{Node}(a, l, r) & \rightarrow t_2 \end{array}$$

where t_1 and t_2 are terms and
the computation rules are:

When $t = \text{Empty}$

(case of(Empty)) =

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

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

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

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

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:

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

There are several other predicates that take on a similar form to Empty? . Leaf? , Root? and Child? are just a few examples, although they can get more complex.

3 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 [23],[35], [34], [5] [28], and [13] 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 $\forall x, y : A. x = y \vee (x \neq y)$. (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 define a canonical element to be 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. [28]

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 $\text{Node}(a, l, r)$ are constructors for tree_{AS} . 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-Löfs type theory has at its foundation the process of 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 .³

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

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

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

We can develop a similar set of formal rules for and tree_{AS} . They are once again exactly our constructors for the type.

Definition Introduction Rules for tree_A

³We make a presumption of dealing only with finite structures at this point as Nuprl is limited to these, or at least finite lists

$$\frac{}{\text{Empty} \in \text{tree}_A}$$

Empty –*introduction*

$$\frac{a : A \quad l : \text{tree}_A \quad r : \text{tree}_A}{\text{Node}(a, l, r) : \text{tree}_A}$$

Node –*introduction*

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 ϕ : Disjoint Unions, Unit, and Cartesian Products. However as we have seen in tree_A s a recursive type can be parameterized by an arbitrary type A . Any type, recursive or not, parameterized by, or constructed from, another type must consider what it means to be equal in the type A 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 A equality determines what we can say about tree_A 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 occurred [5]. 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 [28]. 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 , π_1 , and π_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*. The natural numbers \mathbb{N} are 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 a 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

$$\forall a, b : A. \text{Eq}_A(a, b) \iff a = b \in A$$

A stronger notion is to show that a type has decidable equality, or is discrete.

Definition 2 (discrete) $discrete\ A \stackrel{def}{=} \forall x, y : A. x =_A y \vee \neg(x =_A y)$

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. Thus, a type A is *discrete* if it's equality is decidable. 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.

3.1 Remarks on decidability

Equality on trees is decidable if the underlying type A is discrete.

Lemma 1 $\forall A : \mathbb{U}. discrete\ A \Leftrightarrow discrete\ tree_A$

If the underlying type A is discrete then trees $tree_A$ are discrete since equality can be determined by recursively comparing the trees node by node. Perhaps the other direction is slightly less obvious. We reduce the problem of deciding two elements of the underlying type, $x, y \in A$ to that of a decision over the tree type by constructing two trees with the values in question as their node values and comparing the trees for equality:

$$Node(x, Empty, Empty) =_{tree_A} Node(y, Empty, Empty)$$

The decision procedure for trees can then, by proxy, decide the equality for A

This means that to actually compute equality of trees (or with a tree-membership predicate $x \in_A t$, which we will go into more detail about below), there must be a method of deciding equality in the type A . However, even if A is not a discrete type, evidence for $x \in_A t$ takes the form of indexes to nodes in t where values y where $y =_A x$ are stored. Suppose we come by some evidence for $(x \in_A t)$, call this index i . If A is not discrete, we will not be able to verify that the value stored at the node indexed by i actually contains a value equal to x , though the typing tells us it must be. In the context of a proof, such unverifiable evidence is not uncommon, it naturally arises from assumptions specified as antecedents of implications.

4 Membership in a Tree

The above development of binary trees sets up the environment in which we can begin to describe the connection between membership predicates and indexes into a tree structure. We will continue to use the same tree constructors as above as well as use trees that will store an element of an arbitrary type A at the nodes. The leaves will continue to be empty. Hereafter we will use T_A to designate our trees, forshadowing that these results can be generalized to any recursive type.

4.1 Indexes

An index generally conjures the idea of a natural number that references an element contained in a list. We are speaking more generally about indexes. We think of indexes not as a discrete reference into the structure, but as a path which will lead us through the structure to the point, the node, that holds the information that we require. This can be seen in a translation from the natural number into a description of the path, or index, into the list. We congruently describe an index n into a list as the number of injections (*inr* or *inl*) we must make until we reach the location in the list. The location in a list is a description of which term we will find our element of interest in the nesting of disjunctions. At this point we need an indication that we are exactly at the appropriate location. Since we are dealing with the evidence, or proof, of this fact our evidence will simply be \cdot , or "it" as described above. The last term in this path will be *inr* or *inl* depending on the location. If we were to take the list $[a, b, c, d]$ the element a is indexed by 0 or $inl(\cdot)$. The element c is indexed by 2 or $inr(inr(inl(\cdot)))$ and d corresponds to 3 or $inr(inr(inl(inl(\cdot))))$. Obviously the natural number is a cleaner and more concise representation. However we can build the function that translates from one to the other. Paths into trees are similar. For any recursive type the indexes are unique but the structure of those indexes are increasingly complex in their structure. In what follows we will use the terms path and index interchangeably. We remark here that $outl(inl(x)) = x$ and $outr(inr(x)) = x$.

4.2 Membership $x \in_A t$

A predicate describing membership in some structure, be it lists, trees or some other structure, is an easily programmed and defined function. A simple exercise for any undergraduate, in any language. What we describe below is the natural definition in that it is the membership predicate most every functional programmer confronted with this type would write. It is notable though, and we discuss this idea further in section 6, that there can be nuances within the definition that create different inhabitants or variations in the proof. Membership⁴ in a structure of type T_A can be defined most naturally as follows:

Definition 3 (Membership in a tree $(x \in_A t)$)

$$\begin{aligned} \forall A:\mathbb{U}. \forall x:A. \forall t:T_A. (x \in_A t) &\in \mathbb{P} \\ (x \in_A \text{Empty}) &\stackrel{\text{def}}{=} \text{False} \\ (x \in_A \text{Node}(y, t_1, t_2)) &\stackrel{\text{def}}{=} x =_A y \vee x \in_A t_1 \vee x \in_A t_2 \end{aligned}$$

The first line gives the well-formedness theorem characterizing the type of the membership predicate. It says that for every type A and for every T_A tree t and every element x of A , $(x \in_A t)$ is a proposition. \mathbb{P}_i denotes the propositions at some (polymorphic) level i of

⁴We have rather heavily overloaded the membership symbol. Membership in a type or of a type in a universe should be reasonably easy to distinguish from membership in a tree based on the context. $x \in_A t$ is a defined membership predicate for trees where the type A is a parameter to the predicate indicating which equality to use.

the hierarchy of propositions. In Nuprl, the hierarchy of propositional universes is just the hierarchy of type universes *i.e.* $\mathbb{P}_i \stackrel{\text{def}}{=} \mathbb{U}_i$. We normally write $\mathbb{P}(\mathbb{U})$ for $\mathbb{P}_i(\mathbb{U}_i)$. We noted the hierarchy of universes earlier without explicitly mentioning that they too are indexed. The equality between \mathbb{P} and \mathbb{U} derives from the correspondence between propositions as types, that every type can be viewed as a proposition and vice versa. We can see this in the correspondence between \vee (logical or) and $+$. Since we are working in a constructive setting the disjunction property holds, *i.e.* the proposition $\phi \vee \psi$ holds if at least one of ϕ or ψ holds, *and we know which one*. Thus, the proposition-as-types interpretation indicates that $\phi \vee \psi$ is true if the disjoint union $\phi + \psi$ is inhabited. So, looking back at the definition of membership, inhabitants of $(x \in_A \text{Empty})$ are just the inhabitants of False ⁵ (*i.e.* there are none) and the inhabitants of

$$x =_A y \vee x \in_A t_1 \vee x \in_A t_2$$

are of the form $\text{inl}(u)$ where u inhabits $x =_A y$ or $\text{inr}(\text{inl}(p))$ where p inhabits the type $(x \in_A t_1)$ or are of the form $\text{inr}(\text{inr}(p))$ where p inhabits the type $(x \in_A t_2)$. (We assume disjunction associates to the right.) To take this back to a similar and easily visualized structure, we can begin to see the analogy to lists in the definition on indexes.

Inhabitants are terms that serve as evidence for the truth of an instance of the predicate. If a proposition has a proof, the instances of that proof inhabit the type corresponding to the proposition. So a type is inhabited if and only if it has a proof. We noted early that Void has no inhabitants. The corresponding proposition, False , has no proof. Inhabitants of this membership predicate are terms in a sum over the types $\mathbf{1}$ and $\mathbf{0}$. The shape of the sum term has a structure matching the shape of the tree where internal nodes are translated as type $\mathbf{1}$ and leaves are translated as type $\mathbf{0}$ (since the predicate always returns false on leaves.). *e.g.* the possible inhabitants of the membership in the tree t defined as

$$\text{Node}(a, \text{Node}(b, \text{Empty}, \text{Empty}), \text{Node}(a, \text{Empty}, \text{Empty}))$$

are inhabitants of the type

$$\mathbf{1} + ((\mathbf{1} + (\mathbf{0} + \mathbf{0})) + (\mathbf{1} + (\mathbf{0} + \mathbf{0})))$$

To see this, note that the recursive unfolding of the membership predicate $x \in_A t$ evaluates to the following term.

$$x =_A a \vee (((x =_A b) \vee \text{False} \vee \text{False}) \vee ((x =_A a) \vee \text{False} \vee \text{False}))$$

Since a occurs twice in this tree, evidence for $a \in t$ takes one of two forms $\text{inl}(\cdot)$ or $\text{inr}(\text{inr}(\text{inl}(\cdot)))$ while evidence for $b \in t$ is of the form $\text{inr}(\text{inl}(\text{inl}(\cdot)))$. There are no other inhabitants of this type.

⁵ $\text{False} \stackrel{\text{def}}{=} \mathbf{0}$

4.3 Abstracting Shape

If t is a tree, we denote the *membershape* of t as $\wr t\wr$. Shape can be thought of in several different fashions. The first is that it is the structure of the tree without any content, but a $\mathbf{1}$ indicates where we may possibly have content, a $\mathbf{0}$ indicates where we definitely will not. Another view is that we can take the shape of the tree as the collection of all the possible paths from the root to the leaves. Of course in doing this we describe all of the paths to each node in the tree. We define a mapping from a tree to the sum type representing the shape of the paths to internal nodes as follows:

Definition 4 (member shape)

$$\begin{aligned} \forall A:\mathbb{U}. \forall t:T_A. \wr t\wr &\in \mathbb{U} \\ \wr Empty\wr &\stackrel{def}{=} \mathbf{0} \\ \wr Node(x, t_1, t_2)\wr &\stackrel{def}{=} \mathbf{1} + \wr t_1\wr + \wr t_2\wr \end{aligned}$$

The first statement above gives us the fact that membershape is a type. Note that this type is discrete which we defined earlier. Here $\wr t\wr$ is discrete when, as before, equality on inhabitants of the type $\wr t\wr$ is decidable.

Lemma 2 (membershape discrete) $\forall A:\mathbb{U}. \forall t:T_A. discrete \wr t\wr$

The proof is by induction on the structure of t .

A type is finite iff it is in one-to-one correspondence with some initial prefix of the natural numbers.

Definition 5 (finite) $finite A \stackrel{def}{=} \exists n:\mathbb{N}. A \sim \{0 \dots n\}$

A proof of a type A being finite is a function (specifically a bijection) from the elements of the A onto an initial segment of the natural numbers. We can use this witnessing function then to decide equality between any two elements. This leads then to the fact that any finite type is discrete. We recall, based on our discussion of constructive tree equality, that if A is discrete then our trees are discrete, and vice versa. Similarly, we can enumerate all of the possible indexes into the tree because the trees are defined as least fixedpoints, and hence the type of their indexes is finite. This allows us to prove:

Lemma 3 (membershape finite) $\forall A:\mathbb{U}. \forall t:T_A. finite \wr t\wr$

We, finally, define subtype to allow us to compare the types from membership and membershape.

Definition 6 (subtype) $A \subseteq B \stackrel{def}{=} \forall x:A. x \in B$

Theorem 1 $\forall A:\mathbb{U}. \forall x:A. \forall t:A Tree. (x \in_A t) \subseteq \wr t\wr$

So, the inhabitants of the membership predicate $x \in_A t$ are inhabitants of the membership tree for t . To see that these two types are not isomorphic, note that the converse $(\lambda t \subseteq (x \in_A t))$ does not hold. Consider a tree where all the internal nodes store a value (call it y) that is not equal to x , this term in the disjunction will be $y =_T x$ which will be, isomorphic to $\mathbf{0}$, not $\mathbf{1}$. Thus, membership may overstate the inhabitants of a membership predicate of this form. To reiterate, the shape represents a type. The possible inhabitants of this type are the indexes.

If, for some $x \in A$, a tree t , $t \in T_A$ contains the element x at every node, then these types indeed contain the same inhabitants, *i.e.*

$$(x \in_A t) =_{\mathbb{U}} \lambda t$$

$A =_{\mathbb{U}} B$ indicates the two types, A and B , are equal in the universe \mathbb{U} . A and B being equal as types, implies they have the same inhabitants.

Note that the membership λt does not include indexes to the leaves. The definition for the *Empty* case could be modified but then we would lose the close correlation which allows the equality to hold between the inhabitants of the membership predicate $x \in_A t$ and λt when the nodes of the tree t contain only the element x .

4.4 Predicated Shape

It is often the case that we would like to characterize some property of the tree other than simply the shape, the structure without the information stored within the structure. We can modify the membership predicate that builds our shape type. We refine it to take into account the data stored in the nodes of structure as well as its shape, but to do so we need to know the value being searched for. We redefine membership to be sensitive to these issues by generalizing from an arbitrary individual element of A being searched for in the tree to a predicate that must be satisfied at a node or leaf of the tree. The modified predicate reports the shape of the tree, noting where the predicate is satisfied.

We write $\lambda t \int_{\rho}$ for the type characterizing the shape of the tree t where the predicate ρ holds.

Definition 7 (predicated member shape)

$$\forall A : \mathbb{U}. \forall \rho : T_A \rightarrow \mathbb{P}. \forall t : T_A. \lambda t \int_{\rho} \in \mathbb{U}$$

$$\lambda t \int_{\rho} \stackrel{def}{=} \rho(t) + \text{case } t \text{ of}$$

$$\begin{array}{l} \text{Empty} \rightarrow \mathbf{0} \\ | \text{Node}(x, t_1, t_2) \rightarrow \lambda t_1 \int_{\rho} + \lambda t_2 \int_{\rho} \end{array}$$

As noted before, under the propositions-as-types interpretation, *False* is defined to be the empty type $\mathbf{0}$. Of course, we could have instead made the predicate $\rho : A \rightarrow \mathbb{P}$ but this would not allow indexes to leaves of the structure. We would claim that information is

contained both in the indexes of a structure *and* in values stored at internal nodes. Making the predicate over the tree, (*i.e.* of type $T_A \rightarrow \mathbb{P}$) allows for more paths in the structure to be indexed, including paths to the empty node.

We can provide a simple identity that shows that the definition subsumes shape as given in Def. 4 (which simply depended on the value of the element stored at a node) consider the following predicate which returns true if the node value is x and is false otherwise.

Definition 8 ($x =_A$)

$$\begin{aligned} \forall A:\mathbb{U}. \forall x:A. (x =_A) &\in (T_A \rightarrow \mathbb{P}) \\ (x =_A \text{ Empty}) &\stackrel{\text{def}}{=} \text{False} \\ (x =_A \text{ Node}(y, t_1, t_2)) &\stackrel{\text{def}}{=} x =_A y \end{aligned}$$

Using this predicate, we can establish the identity between the inhabitants of the membership predicate $x \in_A t$ and the membershape $\wr t\}_{x=A}$

Theorem 2 $\forall A:\mathbb{U}. \forall x:A. \forall t:A \text{ Tree}. (x \in_A t) =_{\mathbb{U}} \wr t\}_{x=A}$

The proof is by induction on the structure of the tree t . It may appear that Nuprl's equality between types is extensional, it is not. Computation in a type (including the unfolding of definitions) does not change it *i.e.* subject reduction, or computational equality, is built into Nuprl's type system. Since, by the propositions-as-types interpretation, disjunction (\vee) is just defined to be disjoint union ($+$), after a few steps of computation, these types are indeed seen to be intensionally equal.

4.5 Predicated Membership

In the same way we have generalized membershape, we can characterize membership in a tree more abstractly by applying a predicate on trees instead of simply searching for a particular element of type A . We continue to abuse notation by writing $\rho \in t$ for the type of indexes to the members of the tree t where ρ holds.

Definition 9 (predicated membership ($\rho \in t$))

$$\begin{aligned} \forall A:\mathbb{U}. \forall \rho:T_A \rightarrow \mathbb{P}. \forall t:T_A. (\rho \in t) &\in \mathbb{P} \\ \rho \in t &\stackrel{\text{def}}{=} \rho(t) \vee \text{case } t \text{ of} \\ &\quad \text{Empty} \rightarrow \mathbf{0} \\ &\quad | \text{Node}(y, t_1, t_2) \rightarrow \rho \in t_1 \vee \rho \in t_2 \end{aligned}$$

Now membership and shape are perfectly aligned. This agreement is characterized by the following identity.

Theorem 3 $\forall A:\mathbb{U}. \forall t:T_A. \forall \rho:T_A \rightarrow \mathbb{P}. (\rho \in t) =_{\mathbb{U}} \wr t\}_{\rho}$

This theorem is easily proved by induction on the structure of the tree t . It says these membership types and membershapes are equal types *i.e.* that they have the same inhabitants and that those inhabitants respect the same equality. Thus, a natural characterization of the type of indexes ($\lambda t\}._{\rho}$) is just the same as predicated membership when we look at it as a type. This result is not too surprising as would be evidenced by unrolling the two predicates. In the following, we use the two interchangeably.

4.6 Indexing by inhabitants of the membership predicate

Since $\lambda t\}._{\rho}$ is the type inhabited by indexes into t where ρ holds, we should be able to use them as such. The following definition gives a select function; defined so that for each $i \in \lambda t\}._{\rho}$, $t[i]$ is the subtree of t indexed by i .

Definition 10 (select)

$\forall A:\mathbf{U}. \forall t:T_A. \forall \rho:T_A \rightarrow \mathbb{P}. \forall i:\lambda t\}._{\rho}. t[i] \in T_A$

$$\begin{aligned}
t[i] &\stackrel{\text{def}}{=} \text{case } i \text{ of} \\
&\quad \text{inl}(-) \rightarrow t \\
&\quad | \text{inr}(y) \rightarrow \text{case } t \text{ of} \\
&\quad\quad \text{Empty} \rightarrow \text{any}(y) \\
&\quad\quad | \text{Node}(x, t_1, t_2) \rightarrow \text{case } y \text{ of} \\
&\quad\quad\quad \text{inl}(-) \rightarrow t_1[\text{outl}(\text{outr}(i))] \\
&\quad\quad\quad | \text{inr}(-) \rightarrow t_2[\text{outr}(\text{outr}(i))]
\end{aligned}$$

We note that the term $\text{any}(y)$ is computational content of a proof where $\mathbf{0}$ has been assumed; specifically, if $y \in \mathbf{0}$ then, for every type T , $\text{any}(y) \in T$. So, any maps the paradoxical inhabitant of $\mathbf{0}$ to any type at all. This is equivalent to $\perp AX$ in the sequent calculus, where one will assume \perp or derive *False*. In practice, this behaves like exceptions in ML which take any type. An index of the form $\text{inl}(\dots)$ means “This is the indexed node. You’re there!”. An index of the form $\text{inr}(\dots)$ means, “This is not the node, continue on.”. Continue on means, move left or right down the tree and depends on whether $\text{outr}(\text{inr}(\dots))$ is of the form $\text{inl}(\dots)$ [go left] or is of the form $\text{inr}(\dots)$ [go right]. So, if the index is of the form $\text{inr}(\dots)$ and the tree is *Empty*, it is an exception to try to continue on – the index extends off the end of the tree and is not well-formed. The well-formedness theorem stipulates that the index i comes from the indexes in the shape $\lambda t\}._{\rho}$ and so this is impossible.

We will write $\lambda t\}$ for the shape $\lambda t\}_{\lambda x.True}$ which includes *all* indexes into t (including indexes to leaf nodes).

5 Expressiveness of membership

Now, consider the powerset of $\lambda t\}$ which we write as $\mathbf{2}^{\lambda t\}$. Since the set $\lambda t\}$ is finite, and therefore discrete, the functions in $\mathbf{2}^{\lambda t\} \stackrel{\text{def}}{=} \lambda t\} \rightarrow \mathbf{2}$ are all computable and so this type gives

the analog of the classically defined powerset. So, $\mathbf{2}^{\uparrow t}$ is isomorphic to the collection of all subtypes of indexes into the tree t . In some sense we would like know how many of these index sets are expressible using the methods described so far.

Definition 11 (expressible index set) *A type of indexes s where $s \subseteq \uparrow t$ is expressible iff there exists a predicate $\rho : T_A \rightarrow \mathbb{P}$ such that $s =_{\mathbb{U}} \uparrow t \uparrow_{\rho}$.*

With predicated membership ($\rho \in t$) we can, for example, specify indexes of the leaves of t by a predicate that is true when the tree is *Empty* and is false otherwise: $((\lambda x. x =_{T_A} \text{Empty}) \in t)$. We can specify the collections of all indexes into a tree by the predicate which evaluates to the constant *True*. Similarly, we can express the set of all the internal nodes and many other combinations. However, predicates of type $\rho : T_A \rightarrow \mathbb{P}$ are not sensitive to the context or position of a node in a larger tree. So, index sets that are not extensional in ρ (e.g. index sets that depend on the context of the indexed node within the tree) are not expressible. A simple example of this fact is that no predicate $\rho : T_A \rightarrow \mathbb{P}$ can collect the indexes to every other leaf since; no matter where it is encountered, each leaf (*Empty*) is indistinguishable from every other leaf by ρ . Why is this? ρ simply asks local questions, inquiries about the node in question or the root of the subtree currently in focus. It has no context to say what it has seen before, where it is in the structure, or ability to predict what may be coming.

Obviously, predicates $\rho : T_A \rightarrow \mathbb{P}$ can not distinguish t from t' if they are equal trees, if $t =_{T_A} t'$ then $\rho t =_{\mathbb{P}} \rho t'$.

We would like to be able to characterize every possible subset on indexes into a tree. We are after a kind of completeness of expression that the current methods do not give. We have two approaches: (i.) we identify all common subtrees by a quotient construction thereby turning the graph into a directed acyclic graph (DAG) and, (ii.) by extending the predicates to take both the root and an index to the current location thereby gaining a global view of a node in the context of the entire tree.

5.1 Quotienting T_A by common indexes

Since the predicates cannot distinguish equal trees, the indexes so far considered might be quotiented (see [3] and [16]) if they lead to an identical subtree. We first note that it is easy to prove reflexivity, symmetry, and transitivity of tree equality. This gives us that tree equality is an equivalence relation.

Now, consider the relation which is true if its arguments index equal subtrees.

Definition 12 (\equiv_t)

$$\begin{aligned} \forall A : \mathbb{U}. \forall t : T_A. \forall i, j : \uparrow t. (i \equiv_t j) \in \mathbb{P} \\ i \equiv_t j \stackrel{\text{def}}{=} t[i] =_{T_A} t[j] \end{aligned}$$

This relation is easily seen to be an equivalence relation since type equality on T_A is.

Quotient types (supported in Nuprl) are of the form T/E where T is a type and $E : (T \times T) \rightarrow \mathbb{P}$ is an equivalence relation. The inhabitants of the quotient T/E are the equivalence classes on T induced by E . The equivalence classes are named by the elements of the unquotiented type T , and so each notation for an element of T is a notation for an equivalence class in T/E . Each element (equivalence class) in T/E may have many distinct names, but these names all denote the same elements of the quotient type.

In our case, $\{\!\!|t\}\!\!/ \equiv_t$ identifies indexes of t if the subtrees they index are equal in the type T_A ; equivalently

$$i =_{(\{\!\!|t\}\!\!/ \equiv_t)} j \text{ iff } t[i] =_{T_A} t[j]$$

The quotiented structure indexed in this way is a DAG representation of the tree structure T_A and local predicates are completely expressive with respect to this structure.

By quotienting index sets by \equiv_t , we get full expressiveness, in the sense captured by the following theorem.

Theorem 4

$$\forall A:\mathbb{U}. \text{discrete } A \Rightarrow \forall t:T_A. \forall s:\mathbf{2}^{\{\!\!|t\}\!\!}. \exists \rho:(T_A \rightarrow \mathbb{P}). (s/\equiv_t) =_{\mathbb{U}} (\{\!\!|t\}\!\!/ \equiv_t)$$

Note that for $s \in \mathbf{2}^{\{\!\!|t\}\!\!}$, we will abuse notation by simply writing s for the indexes in the type $\{i : \{\!\!|t\}\!\!|s(i) = 1\}$, (*e.g.* s/\equiv_t is the type $\{i : \{\!\!|t\}\!\!|s(i) = 1\}/\equiv_t$.)

So, the theorem says that given a tree t over a discrete type A , and given a subset of indexes s , there exists a predicate ρ that perfectly characterizes s modulo the equivalence \equiv_t .

The witness for the predicate ρ in the proof is the one that checks if its input (say r) is among the subtrees of t indexed by s . Since this set is finite and since A is discrete, the type T_A is finite and discrete as well, we can just enumerate the trees indexed by s checking if they are equal to r . Now, consider indexes i such that $i \in s$. i is just a name for the equivalence class $\{j \in \{\!\!|t\}\!\!|j \equiv_t i\}$. So the quotient s/\equiv_t (possibly) expands the number of indexes naming its elements. It expands the set of indexes to include those indexes that can not be distinguished by the predicate ρ .

So we see that the quotient essentially grows s to include all the indexes to extensionally equal trees. Hence it is an over statement of the sets but the DAG itself is fully expressible under the notion that any predicate would be unable to differentiate equal subtrees.

5.2 Global Membership Predicates

An alternative to growing the index set s through the quotient construction is to modify the expressiveness of the predicate so that subtrees can be distinguished by their context in the tree being searched. The way to do this is to carry state information through the computation that indicates not only which tree is being evaluated, but its context in the larger tree. Computation in Nuprl can be viewed as a purely functional language and hence state is not part of its mode of operation.

Consider the following dependent type:

$$\forall t:T_A. \lambda t \rightarrow \mathbb{P}$$

Inhabitants of this type have the form $\lambda t. \lambda i. \phi$ where we can prove

$$t:T_A, i:\lambda t \vdash \phi \in \mathbb{P}$$

, we say the judgment is derivable. The first argument $t \in T_A$ is the tree being searched and $i \in \lambda t$ is the index to the node currently being examined and ϕ is a proposition determining whether the the tree $t[i]$ is a “member”.

The idea for the global search is to pass a dependent predicate (say ρ) of this type both the root of the tree being searched (call it t) and the index i to the current node being searched in the tree. The membership predicate will evaluate $\rho(t)(i)$ and union this result with the search performed on t by appropriately extending i to the left and right branches if t is not the *Empty*. In a way, it is like the cartoon of the train rolling the track out in front of itself. The notion of state is implicit as we essentially recompute the predicate on everything seen up to that point plus the extensions to the left and the right.

To implement this plan this we need a way to consistently extend an index.

Definition 13 (Index Composition)

$$\forall A:\mathbb{U}. \forall t:T_A. \forall i:\lambda t. \forall j:\lambda t[i]. i \circ j \in \lambda t$$

$$i \circ t \stackrel{def}{=} \text{case } i \text{ of}$$

$inl(x) \rightarrow$	t
$ inr(x) \rightarrow$	case x of
$inl(y) \rightarrow$	$inr(inl(y \circ t))$
$ inr(z) \rightarrow$	$inr(inr(z \circ t))$

The well-formedness goal for the composition is worth studying. It is proved by induction on the structure of the tree t . Note that if t is *Empty*, the only index in $\lambda Empty$ is $inl(\cdot)$. Since $Empty[inl(\cdot)] = Empty$, the only possible extension of i is j , where $j \in \lambda Empty[inl(\cdot)]$ which is again $inl(\cdot)$. Looking at the code for composition, the first case says that $inl(\cdot) \circ inl(\cdot) = inl(\cdot)$ which indeed is still an index into *Empty*. The argument in the inductive case is rather straightforward.

The following code implements our strategy for the global search using the composition operator to recursively unroll the indexes down through the tree.

Definition 14 (dependent predicated membership $\rho \in \langle t, i \rangle$)

$$\forall A:\mathbb{U}. \forall \rho:(\forall t:T_A. \lambda t \rightarrow \mathbb{P}). \forall t:T_A. \forall i:\lambda t. (\rho \in \langle t, i \rangle) \in \mathbb{P}$$

$$\rho \in \langle t, i \rangle \stackrel{\text{def}}{=} \rho(t)(i) \vee \text{case } t[i] \text{ of} \\
\begin{array}{l}
\text{Empty} \rightarrow \mathbf{0} \\
| \text{Node} \rightarrow \rho \in \langle t, i \circ \text{inr}(\text{inl}(\cdot)) \rangle \vee \rho \in \langle t, i \circ \text{inr}(\text{inr}(\cdot)) \rangle
\end{array}$$

That this membership predicate is completely expressive is stated as follows.

Theorem 5

$$\forall A : \mathbb{U}. \forall t : T_A. \forall s : \mathbf{2}^{\uparrow t}. \exists \rho : (\forall t : T_A. \uparrow t \rightarrow \mathbb{P}). s =_{\mathbb{U}} (\rho \in \langle t, \text{inl}(\cdot) \rangle)$$

To prove it, use the following witness for the existential

$$\lambda t. \lambda i. s(i) = 1$$

Since $s \in \mathbf{2}^{\uparrow t}$, and since $i \in \uparrow t$, $s(i)$ is computable. By this definition $(\lambda t. \lambda i. s(i) = 1)(t)(i)$ will be true whenever the index i is one of the indexes in s and will return \cdot as evidence for that index of s . If $s(i) \neq 1$ then i is not in s .

6 Programming Considerations

Type theory is a programming language in of itself. As we have seen we can define functions on types and compute the result of those functions. The language of type theory is rather strict and cannot be interpreted, or used, loosely. However with its strict nature also comes a tremendous amount of power and information not necessarily anticipated. Here we illustrate two points. The first exemplifies how we must be careful when specifying a function. The second expresses that when we use constructive type theory we often get something for free.

It is significant that a minor change in the Def. 7 changes the form of the indexes. Consider the following slight alternative to $\uparrow t \}_\rho$ that we write as $\overline{\uparrow t} \}_\rho$.

Definition 15 (predicated member shape (alternate))

$$\forall A : \mathbb{U}. \forall \rho : T_A \rightarrow \mathbb{P}. \forall t : T_A. \overline{\uparrow t} \}_\rho \in \mathbb{U}$$

$$\overline{\uparrow t} \}_\rho \stackrel{\text{def}}{=} \rho(t) + \text{case } t \text{ of} \\
\begin{array}{l}
\text{Empty} \rightarrow \rho(\text{Empty}) \\
| \text{Node}(x, t_1, t_2) \rightarrow \overline{\uparrow t_1} \}_\rho + \overline{\uparrow t_2} \}_\rho
\end{array}$$

The indexes under this slightly modified alternative definition are different from the ones in Def. 7. To see this, assume $\rho(t) = \cdot$ for all trees t . Then $\overline{\uparrow \text{Empty}} \}_\rho$ is inhabited by both $\text{inl}(\cdot)$ and $\text{inr}(\text{inl}(\cdot))$ while the only inhabitant of $\uparrow \text{Empty} \}_\rho$ is $\text{inl}(\cdot)$. This would seem to be undesirable in a number of ways. This example illustrates the sensitivity of the evidence

on the form of the definition. Clearly, these two shape types are equivalent propositionally, *i.e.* the following holds:

$$\forall A:\mathbb{U}. \forall \rho:T_A \rightarrow \mathbb{P}. \forall t:T_A. \wr t \wr_\rho \Leftrightarrow \overline{\wr t \wr}_\rho$$

However, they are not intensionally equal and are not even extensionally equal. This issue needs to be explored in more detail. Quotienting the set of indexes, per Section 5 would eliminate this problem. However the problem derives more from the idea that two programmers may have a slight variation in coding styles which will cause such an issue. Although, as we claim, we wrote the natural membership predicate, there is a nuance in the statement of that predicate.

When we began this investigation we were simply looking to define a type of paths into tree structure. In trying to find the best representation for a path, and a way to define them as a type, we realized that the instances we generated were fairly unnatural. They were based on booleans, and some specially defined types. Membership, which had been formulated early on in our formalization of trees, told us on the surface that an element was in fact in the tree. We viewed it in the classical interpretation, as up until then we had not needed the evidence. However we then took a different approach to the system, and looked at it from the standpoint of membership generating a witness, the constructive view. We first realized this in an analogy to lists which we had much more familiarity. Hence as we formulated the types of paths it became readily apparent that they really were synonymous with the witness provided by a membership predicate. They are isomorphic to be exact. Membership now gives us something even stronger, all the possible paths in a tree. This information can then be used to create dependencies on a tree analogous to the natural number that defines the length of a vector.

7 Conclusions and Work in Progress

Trees are a fundamental structure in computation. Membership predicates are a basic tool we use to describe the contents of these structures and indexes are the representation we use to access that content. In this paper we have described some of the fundamentals of type theory as it relates to recursively defined binary trees. Further we have used the Curry-Howard isomorphism to relate membership predicates on these trees to the indexes into them. This relation has a profound impact on how we view the relationship between type theory and the realm of "practical" programming.

Type theory, and the theorem proving environments it supports, is still not ready to become a replacement for the languages mentioned in the introduction which support strong, expressive type systems. We argue that type theory should no longer be viewed simply as an abstract model of real, practical programming languages. It has too long been relegated to the foundations of constructive logic. Constructive Type theory has at its core the Curry-Howard isomorphism. The details described above, and in particular, the pleasant discovery that in using this correspondence we get something for free, continues this journey from abstract model to practicality.

Addressing the issues raised about the sensitive dependence of specifying a predicate is one that will have to be looked into further. It has the potential to cause incongruencies when programming with proofs. However it is also interesting in its own right as a study since such small variations as this have lead to improvements in the complexity of the resulting programs.

Work is currently underway to generalize this notion of membership and indexes to all recursive types. We have the ability to generate membership predicates and describe the shape (*i.e.* define the type of indexes) for recursive types beyond simply the polynomial types. We have developed this algorithms to build analogous predicates for not only the polynomial recursive types but mutually recursive and nested data types as well. We are interested in following up on the work of Fiore [14] to characterize this development. We also endeavour to extend this work beyond the least fixed point types of Nuprl to the greatest fixed point (co-inductive) types.

References

- [1] Michael Abbott. *Categories of Containers*. PhD thesis, University fo Leicester, 2003.
- [2] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, 2005.
- [3] Stuart Allen. *Nuprl Basics*. Cornell University, 2001.
www.cs.cornell.edu/Info/People/sfa/Nuprl/NuprlPrimitives/.
- [4] Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. Indexed containers. Manuscript, available online, February 2006.
- [5] Roland Backhouse, Paul Chisholm, Grant Malcolm, and Erik Saaman. Do-it-yourself type theory. *Form. Asp. Comput.*, 1(1):19–84, 1989.
- [6] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicael Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saibi, and Benjamin Werner. *The Coq Proof Assistant Reference Manual : Version 6.1*. Technical Report RT-0203, INRIA, Rocquencourt, France, 1997.
- [7] James Caldwell. Moving proofs-as-programs into practice. In *Proceedings, 12th IEEE International Conference Automated Software Engineering*, pages 10–17. IEEE Computer Society, 1997.
- [8] James Caldwell. Classical propositional decidability via Nuprl proof extraction. In Jim Grundy and Malcolm Newey, editors, *Theorem Proving In Higer Order Logics*, volume 1479 of *Lecture Notes in Computer Science*, pages 105–122, 1998.

- [9] James Caldwell. Extracting recursion operators in Nuprl’s type-theory. In A. Pettorossi, editor, *Eleventh International Workshop on Logic -based Program Synthesis, LOPSTR-02*, volume 2372 of *LNCS*, pages 124–131. Springer, 2002.
- [10] James Caldwell, Ian Gent, and Judith Underwood. Search algorithms in type theory. *Theoretical Computer Science*, 232(1-2):55–90, Feb. 2000.
- [11] James Caldwell and Josef Pohl. Constructive membership predicates as index types. *Electr. Notes Theor. Comput. Sci.*, 174(7), 2007.
- [12] Felice Cardone and Mario Coppo. Decidability properties of recursive types. In *ICTCS*, pages 242–255, 2003.
- [13] R. L. Constable et. al. *Implementing Mathematics with the Nuprl proof development system*. Prentice-Hall, 1986.
- [14] Marcelo Fiore. Isomorphisms of generic recursive polynomial types. In *POPL ’04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 77–88, New York, NY, USA, 2004. ACM Press.
- [15] Nicola Gambino and Martin Hyland. Wellfounded trees and dependent polynomial functors. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 210–225. Springer, 2003.
- [16] Martin Hofmann. *Extensional Constructs in Intensional Type Theory*. CPHC/BCS Distinguished Dissertations. Springer Verlag, London, 1997.
- [17] Douglas J. Howe. Reasoning about functional programs in Nuprl. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *Lecture Notes in Computer Science*, Berlin, 1993. Springer Verlag.
- [18] C. B. Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995.
- [19] C. B. Jay and J. R. B. Cockett. Shapely types and shape polymorphism. In D. Sannella, editor, *Programming Languages and Systems (ESOP’94)*, LNCS, pages 302–316. Springer, 1994.
- [20] Christoph Kreitz. Building reliable, high-performance networks with the Nuprl proof development system. *Journal of Functional Programming*, 14(1):21–68, 2004.
- [21] Z. Luo and R. Pollack. LEGO proof development system: User’s manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, 1992.
- [22] Zohar Manna and Richard Waldinger. *The logical basis for computer programming: vol. 2, deductive systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

- [23] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Naples, Italy, 1984.
- [24] C. McBride. The derivative of a regular type is its type of one-hole contexts, 2001.
- [25] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.
- [26] Eugenio Moggi, Gianna Bellè, and C. Barry Jay. Monads, shapely functors, and traversals. *Electr. Notes Theor. Comput. Sci.*, 29, 1999.
- [27] Bengt Nordström. The ALF proof editor. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 253–266, Nijmegen, 1993.
- [28] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Lof’s type theory: an introduction*. Clarendon Press, New York, NY, USA, 1990.
- [29] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.
- [30] Iman Hafiz Poernomo, J. N. Crossley, and Martin Wirsing. *Adapting Proofs-as-Programs: The Curry-Howard Protocol (Monographs in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
- [31] James T. Sasaki. *The Extraction and Optimization of Programs from Constructive Proofs*. PhD thesis, Cornell University, 1985.
- [32] Tim Sheard. Languages of the future. *SIGPLAN Not.*, 39(12):119–132, 2004.
- [33] Tim Sheard. Putting Curry-Howard to work. In *Haskell ’05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 74–85, New York, NY, USA, 2005. ACM Press.
- [34] Giovanni Sommaruga. *History and Philosophy of Constructive Type Theory*, volume 290 of *Synthese Library*. Kluwer Academic Pub., 2000.
- [35] Simon Thompson. *Type theory and functional programming*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1991.
- [36] Tjark Weber and James Caldwell. Constructively characterizing fold and unfold, 2003.