Constructive membership predicates as index types

James Caldwell [*]and Josef Pohl [†]
Department of Computer Science
University of Wyoming
Laramie, WY

July 19, 2006

Abstract

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 show that the inhabitants of membership predicates are precisely the inhabitants of a generic shape type. We show that membership of x (of type T) in structure S, $(x \in_T S)$ can not, in general, index all parts of a structure S and we generalize to a form $\rho \in S$ where ρ is a predicate over S. Under this scheme, $(\lambda x.True) \in S$ 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 retain state "from above" as well as allow "looking below". 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

Recently, there has been significant interest in combining techniques of formal constructive proof with programming type systems and language based approaches to verification [28]; witness this workshop. Some of the most notable recent efforts include the Omega programming language[26, 27] and the Epigram programming language[21]. These are two examples of recent programming languages that push the application of the Curry-Howard isomorphism down into the realm of "practical" programming. Similarly, the work on the

[*]The first authors work was partially supported by NSF grant CCR-9985239.
[†]This work was supported by a Wyoming NASA Space Grant Graduate Fellowship

Applied Type System (ATS) [10] looks to incorporate proofs in programs to establish the validity of constraints.

In this paper, we work in Nuprl, [11] a Martin Löf 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 might be considered "old" Curry-Howard technology [1], we apply it here to the problem of programming with proofs. This application of Nuprl is not new [25, 15, 6, 7, 9, 8, 18].

1.1 Membership proofs as 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 $\mathbb{Z}\ \mathit{List}$ 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 setting[2] 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.

[1]Coq [5] and, by now, Lego [19] and Alf[24] are similarly long-in-the-tooth and have been used for similarly rich examples programing via the Curry-Howard isomorphism.

[2]Indeed 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 [17, 16], 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 [23], 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 [13] and polynomial recursive type [12]. We believe that the work described here can be lifted to the more general setting and intend to do so in future work.

We apply constructive type theory to examine the constructive content of membership predicates as indexes.

2 Membership in a Tree

We will motivate the ideas in the context of a type of binary trees storing values of an arbitrary type A in the internal nodes of the tree and whose leaves are empty.

2.1 A binary tree type

We formally define the type of Binary trees as follows:

$$T_A \overset{def}{=} \mu X.\, \mathbf{1} + A \times X \times X$$

where $\mu X.\phi$ is a recursive type, the least fixedpoint of type ϕ, which in this case is the polynomial $\mathbf{1} \ + \ A \times X \times X$. The disjoint union of types A and B is denoted $A + B$ and the product $A \times B$ denotes Cartesian product of A and B. The type $\mathbf{1}$ is the "unit" type having exactly one element, we call that element "it" and display it as "$\cdot$". For completeness of the propositions-as-types interpretation, every proposition must be interpretable as a type, including equality propositions of the form $x =_A y$. This proposition is meaningful only if both x and y are in A. If true (*i.e.* if x and y are equal in A) the type $x =_A y$ is inhabited by *it*, the single element of the unit type and so is isomorphic to $\mathbf{1}$ and; if false, the type is uninhabited, and is isomorphic to the empty type $\mathbf{0}$.

We have the following well-formedness theorem for T_A.

$$\forall A\!:\!\mathbb{U}.\ T_A \in \mathbb{U}$$

It says that for any type, A the type of trees T_A is a type. This type is well-formed because all occurrences of the bound variable X occur in strictly positive positions.

The elements of the type T_A are terms of the form $inl(\cdot)$ (which we will label $Empty$) or are terms of the form $inr(a, t_1, t_2)$ (which we will display as $Node(a, t_1, t_2)$). We remark here that $outl(inl(x)) = x$ and $outr(inr(x)) = x$.

2.2 Membership $x \in_A t$

Membership[3] in a structure of type T_A can be defined most naturally[4] as follows:

Definition 1 (Membership in a tree ($x \in_A t$)).

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

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 [5]. Note that we are working in a constructive setting and so 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 $(x \in_A Empty)$ are just the inhabitants of $False$[6] (*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 $inl(u)$ where u inhabits $x =_A y$ or $inr(inl(p))$ where p inhabits the type $(x \in_A t_1)$ or are of the form $inr(inr(p))$ where p inhabits the type $(x \in_A t_2)$. (We assume disjunction associates to the right.)

Thus, inhabitants of this membership predicate (terms that serve as evidence for the truth of an instance of the predicate) are terms in a sum over the types **1** and **0**. The shape of the sum term has a structure matching the shape of the tree where internal nodes are translated as type **1** and leaves are translated as type **0** (since the predicate always returns false on leaves.). *e.g.* the possible inhabitants of the membership in the tree t defined as

$$Node(a, Node(b, Empty, Empty), Node(a, Empty, Empty))$$

[3]We 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.

[4]By "naturally" we mean that this is the membership predicate most every functional programmer confronted with this type would write.

[5]$\mathbb{P}_i$ denotes the propositions at some (polymorphic) level i of the hierarchy of propositions. In Nuprl, the hierarchy of propositional universes is just the hierarchy of type universes *i.e.* $\mathbb{P}_i \overset{def}{=} \mathbb{U}_i$. We normally write $\mathbb{P}$ ($\mathbb{U}$) for $\mathbb{P}_i$ ($\mathbb{U}_i$).

[6]$False \overset{def}{=} \mathbf{0}$

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.

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

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

2.3 Abstracting Shape

If t is a tree, we denote the *membershape* of t as $\wr t \int$. We define a mapping from a tree to the sum type representing the shape of the paths to internal nodes as follows:

Definition 2 (member shape).

$\forall A{:}\mathbb{U}.\, \forall t{:}T_A.\ \wr t \int \in \mathbb{U}$
$\wr \mathit{Empty} \int \overset{\mathit{def}}{=} \mathbf{0}$
$\wr \mathit{Node}(x, t_1, t_2) \int \overset{\mathit{def}}{=} \mathbf{1} + \wr t_1 \int + \wr t_2 \int$

Note that this type is discrete *i.e.* equality on inhabitants of the type $\wr t \int$ is decidable.

Definition 3 (discrete). $\mathit{discrete}\ A \overset{\mathit{def}}{=} \forall x, y{:}A.\ x =_A y \vee \neg(x =_A y)$

Thus, a type A is *discrete* if it's equality is decidable. Natural numbers are discrete, the type of functions $\mathbb{N} \to \mathbb{N}$ is not.

Lemma 1 (membershape discrete). $\forall A{:}\mathbb{U}.\, \forall t{:}T_A.\ \mathit{discrete}\ \wr t \int$

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

Definition 4 (finite). $\mathit{finite}\ A \overset{\mathit{def}}{=} \exists n{:}\mathbb{N}.\ A \sim \{0 \ldots n\}$

Since trees are defined as least fixedpoints, the type of their indexes is finite.

Lemma 2 (membershape finite). $\forall A{:}\mathbb{U}.\, \forall t{:}T_A.\ \mathit{finite}\ \wr t \int$

The evidence for the finiteness of a type A (a bijection from A to an initial segment of $\mathbb{N}$) gives a method of deciding equality. Thus, all finite types are discrete.

Definition 5 (subtype). $A \subseteq B \overset{\mathit{def}}{=} \forall x{:}A.\ x \in B$

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

So, the inhabitants of the membership predicate $x \in_A t$ are inhabitants of the membershape tree for t. To see that these two types are not isomorphic, note that the converse ($\wr t \int \subseteq (x \in_A t)$) does not hold. Consider a tree where the internal node stores 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, membershape over approximates the inhabitants of a membership predicate of this form.

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.*

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

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

2.3.1 Remarks on decidability

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

Lemma 3. $\forall A{:}\mathbb{U}.\ \mathit{discrete}\, A \Leftrightarrow \mathit{discrete}\, T_A$

If the underlying type A is discrete then trees T_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:

$$\mathit{Node}(x, \mathit{Empty}, \mathit{Empty}) =_{T_A} \mathit{Node}(y, \mathit{Empty}, \mathit{Empty})$$

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

It should be noted that to actually compute equality of trees (or with the tree-membership predicate $x \in_A t$), 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.

2.4 Predicated Shape

We can refine the shape type 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 modify membershape to be sensitive to these issues by generalizing from an 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.

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

Definition 6 (predicated member shape).

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

$$\begin{array}{lll} \wr t \int_\rho & \stackrel{def}{=} & \rho(t)\ +\ \mathit{case}\ t\ \mathit{of} \\ & & \qquad\qquad \mathit{Empty}\ \ \longrightarrow \mathbf{0} \\ & & \qquad\qquad \mid\ \mathit{Node}(x, t_1, t_2) \longrightarrow \wr t_1 \int_\rho\ +\ \wr t_2 \int_\rho \end{array}$$

Note that 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 $\rho : A \to \mathbb{P}$ but this would not allow indexes to leaves of the structure and 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 $T_A \to \mathbb{P}$) allows for more paths in the structure to be indexed, including paths to the empty node.

To see that this definition subsumes shape as given in Def. 2 (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 7 ($x =_A$).

$$\begin{array}{l} \forall A{:}\mathbb{U}.\ \forall x{:}A.\ (x =_A) \in (T_A \to \mathbb{P}) \\ (x =_A \mathit{Empty}) \stackrel{def}{=} \mathit{False} \\ (x =_A \mathit{Node}(y, t_1, t_2)) \stackrel{def}{=} x =_A y \end{array}$$

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

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

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 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 intentionally equal.

It is significant that a minor change in the Def. 6 changes the form of the indexes. Consider the following slight alternative to $\wr t \int_\rho$ that we write as $\overline{\wr t \int}_\rho$.

Definition 8 (predicated member shape (alternate)).

$\forall A : \mathbb{U}.\ \forall \rho{:}T_A \to \mathbb{P}.\ \forall t{:}T_A.\ \overline{\wr t \int}_\rho \in \mathbb{U}$

$$\begin{array}{lll} \overline{\wr t \int}_{\rho} & \overset{def}{=} & \rho(t) \;+\; \mathit{case}\ t\ \mathit{of} \\ & & \qquad \mathit{Empty} \;\longrightarrow \rho(\mathit{Empty}) \\ & & \qquad \mid \mathit{Node}(x, t_1, t_2) \longrightarrow \overline{\wr t_1 \int}_{\rho} \;+\; \overline{\wr t_2 \int}_{\rho} \end{array}$$

The indexes under this slightly modified alternative definition are different from the ones in Def. 6. To see this, assume $\rho(t) = \cdot$ for all trees t. Then $\overline{\wr \mathit{Empty} \int}_{\rho}$ is inhabited by both $\mathit{inl}(\cdot)$ and $\mathit{inr}(\mathit{inl}(\cdot))$ while the only inhabitant of $\wr \mathit{Empty} \int_{\rho}$ is $\mathit{inl}(\cdot)$. 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:

$$\forall A\!:\!\mathbb{U}.\ \forall \rho\!:\!T_A \to \mathbb{P}.\ \forall t\!:\!T_A.\ \wr t \int_{\rho} \Leftrightarrow \overline{\wr t \int}_{\rho}$$

However, they are not intensionally equal and are not even extensionally equal.

2.5 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 ($\rho \in t$)).
$\forall A\!:\!\mathbb{U}.\ \forall \rho\!:\!T_A \to \mathbb{P}.\ \forall t\!:\!T_A.\ (\rho \in t) \in \mathbb{P}$

$$\begin{array}{lll} \rho \in t & \overset{def}{=} & \rho(t) \;\vee\; \mathit{case}\ t\ \mathit{of} \\ & & \qquad \mathit{Empty} \;\longrightarrow \mathbf{0} \\ & & \qquad \mid \mathit{Node}(y, t_1, t_2) \longrightarrow \rho \in t_1 \;\vee\; \rho \in t_2 \end{array}$$

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

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

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 ($\wr t \int_{\rho}$) is just the same as predicated membership when we look at it as a type. In the following, we use the two interchangeably.

2.6 Indexing by inhabitants of the membership predicate

Since $\wr t \int_{\rho}$ 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 $i \in \wr t \int_{\rho}$, $t[i]$ is the subtree of t indexed by i.

Definition 10 (select).

$\forall A{:}\mathbb{U}.\, \forall t{:}T_A.\, \forall \rho{:}T_A \to \mathbb{P}.\, \forall i{:}\wr t \int_\rho.\ \ t[i] \in T_A$

```
$t[i] \overset{def}{=}$ $case\ i\ of$
        $inl(\_) \to t$
      $|\ inr(y) \to case\ t\ of$
                    $Empty \longrightarrow any(y)$
                  $|\ Node(x, t_1, t_2) \longrightarrow case\ y\ of$
                                          $inl(\_) \to t_1[outl(outr(i))]$
                                        $|\ inr(\_) \to t_2[outr(outr(i))]$
```

We note that the term $any(y)$ is computational content of a proof where $\mathbf{0}$ has been assumed; specifically, if $y \in \mathbf{0}$ then, for every type T, $any(y) \in T$. So, *any* maps the paradoxical inhabitant of $\mathbf{0}$ to any type at all. In practice, this behaves like exceptions in ML which take *any* type. An index of the form $inl(\cdots)$ means "This is the indexed node. You're there!". An index of the form $inr(\cdots)$ means, "This is not the node, continue on.". Continue on means, move left or right down the tree and depends on whether $outr(inr(\cdots))$ is of the form $inl(\cdots)$ [go left] or is of the form $inr(\cdots)$ [go right]. So, if the index is of the form $inr(\cdots)$ 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 $\wr t \int_\rho$ and so this is impossible.

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

3 Expressiveness of membership

Now, consider the powerset of $\wr t \int$ which we write as $\mathbf{2}^{\wr t \int}$. Since the set $\wr t \int$ is finite, and therefore discrete, the functions in $\mathbf{2}^{\wr t \int} \overset{def}{=} \wr t \int \to \mathbf{2}$ are all computable and so this type gives the analog of the classically defined powerset. So, $\mathbf{2}^{\wr t \int}$ 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 $s \subseteq \wr t \int$ is* expressible *iff there exists a predicate $\rho : T_A \to \mathbb{P}$ such that $s =_{\mathbb{U}} \wr t \int_\rho$.*

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: $((\lambda x.\, x =_{T_A} Empty) \in t)$. 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 $\rho : T_A \to \mathbb{P}$ 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. For example, no predicate $\rho : T_A \to \mathbb{P}$ 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.

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

We 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.

3.1 Quotienting T_A by common indexes

Since the predicates cannot distinguish equal trees, the indexes so far considered might be quotiented (see [3] and [14]) if they lead to an identical subtree.

Consider the relation which is true if its arguments index equal subtrees.

Definition 12 ($\equiv_t$).

$$\begin{array}{l} \forall A{:}\mathbb{U}.\ \forall t{:}T_A.\ \forall i,j{:}\wr t\int.\ (i \equiv_t j) \in \mathbb{P} \\ i \equiv_t j \ \overset{def}{=}\ \ t[i] =_{T_A} t[j] \end{array}$$

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 $E : (T \times T) \to \mathbb{P}$ 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, $\wr t\int / \equiv_t$ identifies indexes of t if the subtrees they index are equal in the type T_A; equivalently

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

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.

$$\forall A{:}\mathbb{U}.\ \mathit{discrete}\, A \Rightarrow \forall t{:}T_A.\ \forall s{:}\mathbf{2}^{\wr t\int}.\ \exists \rho{:}(T_A \to \mathbb{P}).\ (s/\equiv_t) =_{\mathbb{U}} (\wr t\int_\rho/\equiv_t)$$

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

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 $\{j \in \wr t \int | j \equiv_t i\}$. 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.

3.2 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.

Consider the following dependent type:

$$\forall t{:}T_A.\ \wr t \int \to \mathbb{P}$$

Inhabitants of this type have the form $\lambda t.\lambda i.\phi$ where the judgment

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

is derivable. The first argument $t \in T_A$ is the tree being searched and $i \in \wr t \int$ 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.

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

Definition 13 (Index Composition).

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

$$
\begin{array}{lll}
i \circ t & \overset{def}{=} & case\ i\ of \\
& & \quad inl(x) \rightarrow t \\
& & \mid inr(x) \rightarrow case\ x\ of \\
& & \qquad\qquad inl(y) \rightarrow inr(inl(y \circ t)) \\
& & \qquad\quad \mid inr(z) \rightarrow inr(inr(z \circ t))
\end{array}
$$

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 $\wr Empty \int$ is $inl(\cdot)$. Since $Empty[inl(\cdot)] = Empty$, the only possible extension of i is j, where $j \in \wr Empty[inl(\cdot)] \int$ 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 $\rho \in \langle t, i \rangle$).

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

$$
\begin{array}{lll}
\rho \in \langle t, i \rangle & \overset{def}{=} & \rho(t)(i) \vee case\ t[i]\ of \\
& & \qquad Empty \longrightarrow \mathbf{0} \\
& & \quad \mid Node \longrightarrow \rho \in \langle t, i \circ inr(inl(\cdot)) \rangle \ \vee\ \rho \in \langle t, i \circ inr(inr(\cdot)) \rangle
\end{array}
$$

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

Theorem 5.

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

To prove it, use the following witness for the existential

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

Since $s \in \mathbf{2}^{\wr t \int}$, and since $i \in \wr t \int$, $s(i)$ 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.

4 Conclusions and Future Work

We have used the Curry-Howard isomorphism in deep way to identify membership predicates on trees with indexes into those trees.

The issue that came up with the introduction of Def. 8 – regarding the sensitivity of the types of the inhabitants to the formulation of the membership or shape predicate – is potentially a serious issue that needs to be explored more. We could use a quotient type, as we did in Section 3 to mitigate a problem like multiple indexes for one subtree – though we have not done it. In general, the Curry-Howard based evidence is rather delicate and dependent on the language used to generate it. This sensitivity to seemingly small differences in the form an expression may take seems to violate general principles of robustness that is the basis for much software engineering practice. In the end, getting indexes for free from easily specified membership predicates may outweigh any possible drawbacks based on the sensitivity argument. These issues need to be investigated more thoroughly.

We are interested in seeing how the techniques of [20] for data structure transformation might be applied to translations between index types.

As an alternative to the fully expressive membership predicate presented in Def. 14 we are considering formalizing a tree based state monad [29, 22] to pass the global state through the computation.

We do not see any significant technical obstacles to generalizing the results presented in this paper for binary trees to arbitrary polynomial recursive types as presented in [12]. We can imagine extending our abstract data type package for Nuprl to uniformly generate membership and shape types for arbitrary recursive types. Certain that it is related to our efforts here, we are trying to decipher the recent work on indexed containers [4]. Also, although Nuprl's recursive types are least fixedpoints, the index type defined here is not obviously incompatible with coinductive types. We plan to explore this issue in future work.

The authors would like to thank the anonymous referees for their comments.

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] 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.

[6] James Caldwell. Moving proofs-as-programs into practice. In *Proceedings, 12th IEEE International Conference Automated Software Engineering*, pages 10–17. IEEE Computer Society, 1997.

[7] 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.

[8] 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.

[9] James Caldwell, Ian Gent, and Judith Underwood. Search algorithms in type theory. *Theoretical Computer Science*, 232(1-2):55–90, Feb. 2000.

[10] Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. *SIGPLAN Not.*, 40(9):66–77, 2005.

[11] R. L. Constable et. al. *Implementing Mathematics with the Nuprl proof development system.* Prentice-Hall, 1986.

[12] 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.

[13] 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.

[14] Martin Hofmann. *Extensional Constructs in Intensional Type Theory.* CPHC/BCS Distinguished Dissertations. Springer Verlag, London, 1997.

[15] 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.

[16] C. B. Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995.

[17] 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.

[18] Christoph Kreitz. Building reliable, high-performance networks with the Nuprl proof development system. *Journal of Functional Programming*, 14(1):21–68, 2004.

[19] Z. Luo and R. Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, 1992.

[20] Nicolas Magaud and Yves Bertot. Changing data structures in type theory: A study of natural numbers. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *TYPES*, volume 2277 of *Lecture Notes in Computer Science*, pages 181–196. Springer, 2000.

[21] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[22] Erik Meijer and Johan Jeuring. Merging monads and folds for functional programming. In J. Jeuring and E. Meijer, editors, *Tutorial Text 1st Int. Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, 24–30 May 1995*, volume 925, pages 228–266. Springer-Verlag, Berlin, 1995.

[23] Eugenio Moggi, Gianna Bellè, and C. Barry Jay. Monads, shapely functors, and traversals. *Electr. Notes Theor. Comput. Sci.*, 29, 1999.

[24] Bengt Nordström. The ALF proof editor. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 253–266, Nijmegen, 1993.

[25] James T. Sasaki. *The Extraction and Optimization of Programs from Constructive Proofs.* PhD thesis, Cornell University, 1985.

[26] Tim Sheard. Languages of the future. *SIGPLAN Not.*, 39(12):119–132, 2004.

[27] 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.

[28] A. Stump. Programming with proofs: Language-based approaches to totally correct software. In N. Shankar, editor, *Verified Software: Theories, Tools, Experiments*, 2005.

[29] Philip Wadler. The essence of functional programming. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, New York, NY, USA, 1992. ACM Press.