A Broader Class of Trees for Recursive Type Definitions for HOL

Elsa L. Gunter

AT&T Bell Laboratories, Rm.#2A-432
600 Mountain Ave., Murray Hill, N.J. 07974, USA
elsa@research.att.com

Abstract. In this paper we describe the construction in HOL of the inductive type of arbitrarily branching labeled trees. Such a type is characterized by an initiality theorem similar to that for finitely branching labeled trees. We discuss how to use this type to extend the system of simple recursive type specifications automatically definable in HOL to ones including a limited class of functional arguments. The work discussed here is a part of a larger project to expand the recursive types package of HOL which is nearing completion. All work described in this paper has been completed.

1 A Broader Class of Recursive Type Definitions

The work described in this paper forms the foundation of a project to expand the class of recursive type specifications for which HOL is capable of automatically defining the types specified and proving the initiality theorem, which acts as an axiomatization for the defined types. The full class of specifications the project aims to handle are those BNF-style specification of the form

$$\begin{aligned} rty_1 &::= C_{1,1}\, ty_{1,1,1} \dots ty_{1,1,k_{1,1}} \mid \dots \mid C_{1,m_1}\, ty_{1,m_1,1} \dots ty_{1,m_1,k_{1,m_1}} \\ &\vdots \\ rty_n &::= C_{n,1}\, ty_{n,1,1} \dots ty_{n,1,k_{n,1}} \mid \dots \mid C_{n,m_n}\, ty_{n,m_n,1} \dots ty_{n,m_n,k_{n,m_n}} \end{aligned}$$

where each type description $ty_{i,j,k}$ must be *admissible*, as defined below (and where we can show that every type specified is well-founded, or in essence, has a base case).

Definition: A type description *ty* is *admissible* (in a given mutually recursive type specification) if it satisfies one of the following three conditions:

- *ty* is an existing type.
- *ty* is rty_i for some i, $1 \leq i \leq n$
- *ty* is of the form $ety \rightarrow tyd$ where *ety* is an existing type and *tyd* is an admissible type description.

It is also possible to extend the notion of admissibility to include occurrences of certain kinds of type constructors, but the precise definition of this case is quite complicated and we omit it here.

This project is composed of three major aspects. The first is the development of a theory of a broader class of (broader) trees in HOL to form the basic building blocks for all other types defined by specifications of the kind described above. The second is the construction of types for a simplified subclass of specifications. The third aspect is the translation between the full class of specifications and the simplified subclass.

The specifications of the simplified subclass are those of the form

$$\mathit{rty} \ ::= \ C_1\,\mathit{ty}_{1,1}\ldots \mathit{ty}_{1,k_1} \mid \ldots \mid C_n\,\mathit{ty}_{n,1}\ldots \mathit{ty}_{n,k_n}$$

where $\mathit{ty}_{i,j}$ is either an existing type, or of the form $\mathit{ty} \rightarrow \mathit{rty}$ for some existing type ty. To see that this simplified class is already a generalization of specifications that are currently handled, notice that any type rty is isomorphic to the type $\mathsf{one} \rightarrow \mathit{rty}$.

The first two aspects of the project have been completed and are discussed in this paper in some detail. The third aspect will only be discussed briefly in the concluding section.

2 Broadening Trees

The current system for automatic definition of recursive types from specifications is founded upon the type of finitely branching labeled trees of finite height. (See [2]). This type will not do as the foundation for the class of specifications which we are attempting to handle. To see this, consider the following specification:

$$\mathsf{tree} ::= \mathsf{Leaf\ integer} \ \mid \ \mathsf{Node\ integer\ (num} \rightarrow \mathsf{tree)}$$

and the following specific tree of that type:

$$\mathsf{flat} = \mathsf{Node\ (neg\ (INT\ 1))}\ (\lambda n.\mathsf{Leaf\ (INT}n))$$

Any finitely branching tree of finite height can have only finitely many leaves, but this tree has a countably infinite set of leaves (one for each natural number). The natural structure to use to model this specification is the collection of arbitrarily branching trees. To be a bit more precise, we will create a polymorphic type of tree, parametrized by a type α, which will act as the indexing set for the branching of trees, and by a type β for labeling the nodes. The type tree described above could be modeled using num for the branching indexing set and integer for the labeling set.

2.1 Partial Functions

In defining our type of broader trees, and providing a succinct "axiomatization" for it, we will want a theory of *partial functions* between two types. As an example of why we would expect to use partial functions in defining this new type of trees, consider the collection of all trees labeled by strings and having at most continuum many subtrees at each node (that is, each node has no more subtrees than there are real numbers). Then, the collection of subtrees at any node of a tree of this type is a set of trees, again of this type, indexed by a subset of the reals. That is, it is a partial function from the reals to trees of this type.

HOL is equipped with a notion of total function, but no built-in notion of partial function. So the question is, how do we best encode the notion of a partial function, given only total functions. To answer this, we define a type constructor lift that is the solution to the specification

$$\alpha\ \mathsf{lift} ::= \mathsf{lift}\ \alpha \mid \mathsf{undefined}$$

The type α lift is characterized by the initiality theorem

$$\forall f e.\ \exists! fn.\ (\forall x.\ fn(\mathsf{lift}\ x) = f\ x) \wedge (fn\ \mathsf{undefined} = e).$$

The constructor lift is one-to-one and has a left inverse lower.

By a partial function from α to β we mean a function from α to β lift. Total functions are injected into partial functions in the obvious manner by the constant $\mathsf{lift\_fun} : (\alpha \to \beta) \to (\alpha \to \beta\ \mathsf{lift})$. The constant

$$\mathsf{lift\_compose} : (\beta \to \gamma\ \mathsf{lift}) \to (\alpha \to \beta\ \mathsf{lift}) \to (\alpha \to \gamma\ \mathsf{lift})$$

is the expected composition of partial functions. The domain of definition of a partial function (*i.e.* the set of values on which the partial function takes on a value other than undefined) is given by the predicate

$$\mathsf{part\_fun\_domain} : (\alpha \to \beta\ \mathsf{lift}) \to \alpha \to \ \mathsf{bool}.$$

The range of a partial function is given by

$$\mathsf{part\_fun\_range} : (\alpha \to \beta\ \mathsf{lift}) \to \beta \to \ \mathsf{bool}.$$

2.2 Broad Trees

Using the type lift, and the constants associated with it, we can now say more precisely what we mean by the type of α-branching β-labeled trees. We mean any polymorphic type $(\alpha, \beta)\tau$ such that the following initiality theorem holds:

$$\begin{array}{l} \exists Node : \beta \to (\alpha \to (\alpha,\beta)\tau\ \mathsf{lift}) \to (\alpha,\beta)\tau. \\ \quad \forall Cases.\ \exists! f.\ \forall\ label\ subtrees. \\ \qquad\qquad f(Node\ label\ subtrees) = \\ \qquad\qquad\quad Cases\ (\mathsf{lift\_compose}\ (\mathsf{lift\_fun}\ f)\ subtrees)\ label\ subtrees \end{array} \tag{1}$$

The value *subtrees* is the indexed set of immediate subtrees of our tree, and *label* is the label at the root node. The term $\mathsf{lift\_compose}\ (\mathsf{lift\_fun}\ f)\ \mathit{subtrees}$ is the set of recursive values of f on the immediate subtrees of the tree, indexed in the same manner over α as the subtrees are. At first glance, this may strike some as a peculiar statement of initiality since there is no obvious separate base case. To see that there is in fact a base case, note that if *subtrees* is the everywhere undefined function, then so is $\mathsf{lift\_compose}\ (\mathsf{lift\_fun}\ f)\ \mathit{subtrees}$, and hence there are no recursive calls for such trees.

Before we begin the discussion of the construction of such trees, let us note a couple of facts about them. Firstly, it is inevitable that we will have to find a model for such trees, if we wish to be able to handle all (well-founded) specifications of the sort described in Section 1, since the type of α-branching β-labeled trees can be described by

$$\mathsf{bonsai} ::= \mathsf{bonsai\_NODE}\ \beta\ (\alpha \to \mathsf{bonsai\ lift})$$

This is an allowable specification, and as we noted above, it has a base case. Therefore, what ever mechanism we devise for handling our class of specifications, it will have to be able to generate a model for this type of α-branching β-labeled trees.

Secondly, these trees behave a bit differently than the finitely branching trees used as the foundation for the current recursive type package in HOL. Not only are these trees potentially infinitely branching, but they potentially have infinite height. It is possible to define a function over these trees using the initiality theorem (1) which returns twice the height of the tree, if the tree has finite height, and returns thirteen otherwise. (See Appendix A.1 for details of the example.) When this function is applied to a tree of infinite height (which can exist), there is no finite subtree for which this function will return the same result. Moreover, no finite number of unwindings of the recursive equation will allow us to eliminate the recursion and directly compute the answer. In the domain theoretic sense, this function is not the limit of its finite approximates. This is radically different behavior than is had by the finitely branching trees. There, every function given by a primitive recursive equation, when applied to a specific tree, can be directly computed by unwinding the recursion as many times as the height of the tree. This fact is central to the proof of existence of functions given by primitive recursion. We do not have any such fact available to us, and hence we shall have to take a different route entirely to show existence.

To begin the construction of our broader trees, we will first build the unlabeled kind. An α-branching unlabeled tree is represented by the set (described by a predicate) of its finite branches (where a branch always starts at the root). A finite branch is a list of αs, describing which index was selected at each height. A set of finite branches is the branch set of an α-branching unlabeled tree provided that the empty list (or trivial branch) is in the set and, if a branch is in the set, then all prefixes of the branch are again branches and thus are in the set. Hence, we have the following definition:

$$\begin{array}{l}\forall\ \mathit{branch\_set}\!:\!\alpha\ \mathsf{list} \rightarrow\ \mathsf{bool}.\\ \quad \mathsf{Is\_unlabeled\_tree}\ \mathit{branch\_set} =\\ \quad\quad \mathit{branch\_set}\ []\ \wedge\\ \quad\quad (\forall\ b_1\ b_2.\ \mathit{branch\_set}\ (\mathsf{APPEND}\ b_1\ b_2) \implies \mathit{branch\_set}\ b_1)\end{array}$$

A labeling of an unlabeled tree is a partial function mapping the nodes of the tree to labels. A node is given by the path (or branch) from the root to it. Thus, if the type of the labels is β, a labeling is a partial function from α list to β whose domain of definition is an unlabeled tree.

$$\forall\ l\!:\!\alpha\ \mathsf{list} \rightarrow \beta\ \mathsf{lift}.\ \mathsf{Is\_labeling}\ l = \mathsf{Is\_unlabeled\_tree}(\mathsf{part\_fun\_domain}\ l)$$

At this point, we can define a type of α-branching β-labeled trees, by identifying them with partial functions that are labelings. We will call this type (α,β)broad_tree. Thus we have

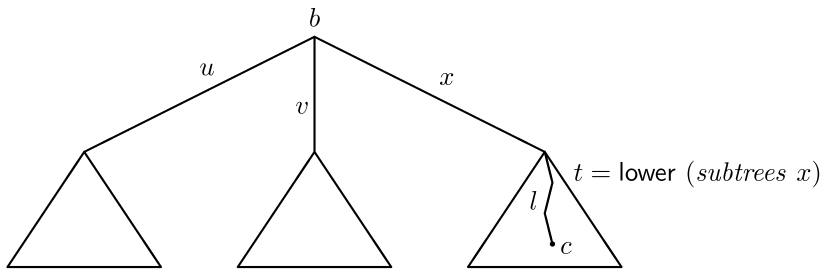
$$\exists \mathit{rep}\!:\!(\alpha,\beta)\mathsf{broad\_tree} \rightarrow (\alpha\ \mathsf{list} \rightarrow \beta\ \mathsf{lift}).\ \mathsf{TYPE\_DEFINITION}\ \mathsf{Is\_labeling}\ \mathit{rep}$$

In the usual manner, we can define the representation and abstraction functions giving the isomorphism between the new type (α,β)broad_tree and the set described by Is_labeling:

$$\begin{array}{l}(\forall a.\ \mathsf{broad\_tree\_ABS}\ (\mathsf{broad\_tree\_REP}\ a) = a)\ \wedge\\ (\forall r.\ \mathsf{Is\_labeling}\ r = \mathsf{broad\_tree\_REP}\ (\mathsf{broad\_tree\_ABS}\ r) = r)\end{array}$$

For more information on defining new types in HOL, see [1, 2].

The next thing we want to do is define a function broad_tree_NODE that will behave as a constructor for terms of type (α,β)broad_tree. Given a function *subtrees* supplying the subtrees of a tree, and a label *label* for the root of the tree, how do we reconstruct the tree? To determine this, we need to know what the set of branches of the new tree is, and what the new labeling is. Each branch of the new tree is either the trivial branch (given by the empty list) or is an element of type α followed by a branch of the tree indexed by that element. Thus, if $\mathit{subtrees}\ x = \mathsf{lift}\ t$ and b is a branch of t, then $x :: b$ is a branch of broad_tree_NODE *label subtrees*. A labeling on this set of branches returns the root label *label* on the trivial branch, and on a nontrivial branch returns the label of the node found at the end of the branch of the subtree indexed by the head of the given branch, where the branch of the subtree is described by the tail of the given branch. Hence, if $\mathit{subtrees}\ x = \mathsf{lift}\ t$ and the branch l of t is labeled by c, then c is the labeling of $x :: l$ in the tree broad_tree_NODE b *subtrees*.



Formally, the definition of $\mathsf{broad\_tree\_NODE}$ is

$$\begin{aligned}
&\forall (label\!:\!\beta)\ (subtrees\!:\!\alpha \to (\alpha,\beta)\mathsf{broad\_tree}).\\
&\quad \mathsf{broad\_tree\_NODE}\ label\ subtrees =\\
&\qquad \mathsf{broad\_tree\_ABS}(\lambda l.\,(l = []) \Rightarrow (\mathsf{lift}\ label)\\
&\qquad\qquad\qquad\qquad\qquad \mid ((subtrees\ (\mathsf{HD}\ l) = \mathsf{undefined})\ \Rightarrow\ \mathsf{undefined}\\
&\qquad\qquad\qquad\qquad\qquad \mid (\mathsf{broad\_tree\_REP}\ (\mathsf{lower}\ (subtrees\ (\mathsf{HD}\ l)))\ (\mathsf{TL}\ l))))
\end{aligned}$$

At this point, it would seem that we are almost done. All we have to do is prove that the initiality theorem holds for $\mathsf{broad\_tree\_NODE}$, right? Unfortunately, life is not so simple. The initiality theorem (1) is false for $\mathsf{broad\_tree\_NODE}$. Both existence and uniqueness fail. (What we have built so far is not the initial algebra, but the final algebra, instead.) For an example of this failure, see Appendix A.2. Since uniqueness fails and induction implies uniqueness, induction must fail also.

2.3 Bonsai

To remedy the failing of induction, we do the standard construction, basically the same construction that allowed us to build the natural numbers from the type of individuals. Let $\mathsf{Is\_bonsai}$ be the intersection of all predicates on $(\alpha,\beta)\mathsf{broad\_tree}$ that are closed under $\mathsf{broad\_tree\_NODE}$. More precisely,

$$\begin{aligned}
&\forall tr.\ \mathsf{Is\_bonsai}\ tr =\\
&\quad (\forall P.\ (\forall subtrees\ label.\ (\forall sbtr.\ \mathsf{part\_fun\_range}\ subtrees\ sbtr \Longrightarrow P\ sbtr) \Longrightarrow\\
&\qquad\qquad P\ (\mathsf{broad\_tree\_NODE}\ label\ subtrees)) \Longrightarrow\\
&\qquad P\ tr)
\end{aligned}$$

In the same way as with the definition of the naturals, we immediately get an induction principle from this construct:

$$\begin{aligned}
&\forall P.\ (\forall subtrees\ label.\ (\forall sbtr.\ \mathsf{part\_fun\_range}\ subtrees\ sbtr \Longrightarrow P\ sbtr) \Longrightarrow\\
&\qquad\quad P\ (\mathsf{broad\_tree\_NODE}\ label\ subtrees)) \Longrightarrow\\
&\quad (\forall tr.\ \mathsf{Is\_bonsai}\ tr \Longrightarrow P\ tr)
\end{aligned} \tag{2}$$

Using the predicate $\mathsf{Is\_bonsai}$, we can now introduce a new type $(\alpha,\beta)\mathsf{bonsai}$ that is in one-to-one correspondence with the set described by $\mathsf{Is\_bonsai}$. (The name bonsai was chosen, in part not to conflict with the names of existing types of trees in HOL, but also in part because the set of bonsai is the subset of broad_tree consisting precisely of those trees having only finite branches. That is, only the broad_trees with short branches are bonsai.) We can also pull over broad_tree_NODE to the type bonsai to get a node constructor bonsai_NODE:

$$\begin{aligned}&\forall label\ subtrees.\ \mathsf{bonsai\_NODE}\ label\ subtrees = \\ &\quad \mathsf{bonsai\_ABS}\,(\mathsf{broad\_tree\_NODE} \\ &\qquad\qquad label \\ &\qquad\qquad (\mathsf{lift\_compose}\ (\mathsf{lift\_fun}\ \mathsf{bonsai\_REP})\ subtrees))\end{aligned}$$

It follows easily from this definition that bonsai_NODE is one-to-one, since broad_tree_NODE is.

It is not so immediate, however, that bonsai_NODE is onto. What we know immediately is only that every broad_tree satisfying Is_bonsai is a node, all of whose subtrees are broad_trees. We don't know off hand that the subtrees of a tree in Is_bonsai are again in Is_bonsai. This fact is proved using the induction principle (2) for Is_bonsai. From the fact that all subtrees of a tree in Is_bonsai are again in Is_bonsai, it follows straight-forwardly that bonsai_NODE is onto. Using the induction principle for Is_bonsai, together with the fact that bonsai_NODE is onto, we can then derive the following induction principle for the type bonsai:

$$\begin{aligned}\forall P.\ &(\forall subtrees\ label.\ (\forall sbtr.\ \mathsf{part\_fun\_range}\ subtrees\ sbtr \Longrightarrow P\ sbtr) \Longrightarrow \\ &\qquad P\ (\mathsf{bonsai\_NODE}\ label\ subtrees)) \Longrightarrow \\ &(\forall tr.\ P\ tr)\end{aligned} \tag{3}$$

2.4 Proving Initiality

Now we are back to trying to prove the initiality theorem (1) again. This time, we know we have an induction principle for the type bonsai and the constructor bonsai_NODE. The uniqueness of functions defined by structural induction over bonsai_NODE follows immediately from our induction principle. Therefore, all we need to show is existence. In previous work with finite trees, the existence of functions defined by structural induction was shown using the heights of the trees. In essence, it could be shown that for a tree of height n, it sufficed to unwind the recursion n times to be able to compute the value of the function on the tree without further recursive calls. As discussed above, this approach will not work in our setting because our trees will not in general have finite height. Another approach must be sought.

The approach we take to demonstrating the existence of such functions is a rather set-theoretic approach: to show such functions exist, we demonstrate a graph which is the graph of such a function. By a graph we mean a relation $g : (\alpha, \beta)\mathsf{bonsai} \to \gamma \to \mathsf{bool}$. Given a case function

$$Cases : (\alpha \to \gamma\ \mathsf{lift}) \to \beta \to (\alpha \to (\alpha, \beta)\mathsf{bonsai}\ \mathsf{lift}) \to \gamma$$

we need to find a relation on (α, β)bonsai and γ that is closed under *Cases* and that is functional. We define what it means for a relation to be closed under a case function as follows:

$$\begin{array}{l}\forall \mathit{Cases}\ \mathit{fun\_rel}.\ \textsf{rel\_is\_case\_closed}\ \mathit{Cases}\ \mathit{fun\_rel} = \\ \quad (\forall \mathit{subtrees}\ \mathit{label}\ \mathit{rec\_fun}. \\ \qquad (\forall x.((\mathit{subtrees}\ x = \textsf{undefined}) = (\mathit{rec\_fun}\ x = \textsf{undefined})) \wedge \\ \qquad\quad (\neg(\mathit{subtrees}\ x = \textsf{undefined}) \Longrightarrow \\ \qquad\quad\ \mathit{fun\_rel}\ (\textsf{lower}(\mathit{subtrees}\ x))\ (\textsf{lower}\ (\mathit{rec\_fun}\ x)))) \Longrightarrow \\ \quad \mathit{fun\_rel}\ (\textsf{bonsai\_NODE}\ \mathit{label}\ \mathit{subtrees})\ (\mathit{Cases}\ \mathit{rec\_fun}\ \mathit{label}\ \mathit{subtrees}))\end{array}$$

The graph we are looking for is the smallest graph that is closed under *Cases*. That is, it is the intersection of all graphs that are closed under *Cases*.

$$\begin{array}{l}\forall \mathit{Cases}\ \mathit{tr}\ z.\ \textsf{smallest\_bonsai\_fun\_rel}\ \mathit{Cases}\ \mathit{tr}\ z = \\ \quad (\forall \mathit{fun\_rel}.\ \textsf{rel\_is\_case\_closed}\ \mathit{Cases}\ \mathit{fun\_rel} \Longrightarrow \mathit{fun\_rel}\ \mathit{tr}\ z)\end{array}$$

As was the case with our definition of Is_bonsai, it follows fairly immediately that smallest_bonsai_fun_rel *Cases* satisfies rel_is_case_closed *Cases*, and that we have the following induction principle:

$$\begin{array}{l}\forall \mathit{Cases}\ \mathit{fun\_rel}.\ \textsf{rel\_is\_case\_closed}\ \mathit{Cases}\ \mathit{fun\_rel} \Longrightarrow \\ \quad (\forall \mathit{tr}\ z.\ \textsf{smallest\_bonsai\_fun\_rel}\ \mathit{Cases}\ \mathit{tr}\ z \Longrightarrow \mathit{fun\_rel}\ \mathit{tr}\ z)\end{array} \tag{4}$$

The fact that smallest_bonsai_fun_rel *Cases* satisfies rel_is_case_closed *Cases* gets us that the function f described by the graph smallest_bonsai_fun_rel *Cases* satisfies the existence half of the initiality theorem, namely that

$$\begin{array}{l}\forall \mathit{subtrees}\ \mathit{label}.\ f(\mathit{Node}\ \mathit{label}\ \mathit{subtrees}) = \\ \quad \mathit{Cases}\ (\textsf{lift\_compose}\ (\textsf{lift\_fun}\ f)\ \mathit{subtrees})\ \mathit{label}\ \mathit{subtrees}\end{array}$$

assuming we know that smallest_bonsai_fun_rel *Cases* describes a function.

To prove that smallest_bonsai_fun_rel *Cases* describes a function, we need to show two things. We need to show that it describes a partial function:

$$\begin{array}{l}\forall \mathit{tr}\ z_1\ z_1.\ (\textsf{smallest\_bonsai\_fun\_rel}\ \mathit{Cases}\ \mathit{tr}\ z_1 \wedge \\ \qquad\qquad \textsf{smallest\_bonsai\_fun\_rel}\ \mathit{Cases}\ \mathit{tr}\ z_2) \Longrightarrow\ (z_1 = z_2)\end{array}$$

and we need to show that it is total in its first argument:

$$\forall \mathit{tr}.\ \exists z.\ \textsf{smallest\_bonsai\_fun\_rel}\ \mathit{Cases}\ \mathit{tr}\ z$$

This latter fact follows by induction on bonsai (3) using the fact that smallest_bonsai_fun_rel *Cases* is closed under *Cases*. The former fact is a bit more involved. Its proof used both induction on bonsai (3) and the induction principle for smallest_bonsai_fun_rel (4). This is the last of the pieces required to get us the initiality theorem we have been seeking:

$$\begin{array}{l}\forall \mathit{Cases}.\ \exists! f.\ \forall \mathit{subtrees}\ \mathit{label}.\ f(\textsf{bonsai\_NODE}\ \mathit{label}\ \mathit{subtrees}) = \\ \qquad\qquad \mathit{Cases}\ (\textsf{lift\_compose}\ (\textsf{lift\_fun}\ f)\ \mathit{subtrees})\ \mathit{label}\ \mathit{subtrees}\end{array}$$

3 A Broader Class of Simple Recursive Types

The next step toward supporting our broader class of recursive type definitions is to handle the simple recursive case. In this section we show how to solve recursive type specifications of the form

$$rty \ ::= \ C_1 \, ty_{1,1} \ldots ty_{1,k_1} \mid \ldots \mid C_n \, ty_{n,1} \ldots ty_{n,k_n}$$

where $ty_{i,j}$ is either an existing type, or of the form $ty \rightarrow rty$ for some existing type *ty*. Given such a specification, a solution for it is a new type *rty*, constructors $C_i : ty_{i,1} \rightarrow \ldots \rightarrow ty_{i,k_i} \rightarrow rty$ and an initiality theorem analogous to (but more complicated than) the one for bonsai. Thus, we need to identify a type in which we can build a model for our specification, define a predicate on that type identifying the elements of the model, and introduce a new type that is isomorphic to the model. Then we need to define the constructors C_i and we need to prove the initiality theorem. In the description that follows, we will often resort to giving examples for each of these steps, rather than giving a completely rigorous description.

3.1 Building the Type

The background type that we are going to use to solve the specification is (σ, τ)bonsai, for some branching type σ and some labeling type τ. The branching type σ and the labeling type τ are each a sum type having one component for each case in the specification. The contribution of a particular case $C_i ty_{i,1} \ldots ty_{i,k_i}$ to the branching type is sum of all $ety_{i,j}$ where $ty_{i,j} = ety_{i,j} \rightarrow rty$, or one if none such exist. The contribution to the labeling type is the product of each $ty_{i,j}$ that is an existing type, if there are any, and one elsewise. For example, the specification

$$\mathsf{toto}{::=}\mathsf{A} \ \mathsf{bool} \ \mathsf{num} \mid \mathsf{B} \ (\alpha \rightarrow \mathsf{toto}) \ (\mathsf{ind} \rightarrow \mathsf{toto}) \mid \mathsf{C} \mid \mathsf{D} \ \mathsf{bool} \ (\mathsf{num} \rightarrow \mathsf{toto})$$

is modeled using the background type

$$(\mathsf{one} + (\alpha + \mathsf{ind}) + \mathsf{one} + \mathsf{num}, (\mathsf{bool} \times \mathsf{num}) + \mathsf{one} + \mathsf{one} + \mathsf{bool}) \ \mathsf{bonsai}$$

Suppose that tr = bonsai_NODE *label subtrees* is a bonsai of the background type. Then *label* is uniquely in one of the summands, say the *i*'th summand, of the labeling type. The predicate that describes the subset of the background type that models the specification is true of *tr* provided that the domain of definition of *subtrees* is either exactly the *i*'th summand of the branching type, if the *i*'th case has a type argument of the form $ty_{i,j} = ety_{i,j} \rightarrow rty$, or is empty otherwise.

Thus, for our example this becomes

$$\begin{array}{l}
\lambda tr.\forall label\ subtrees.(tr = \mathsf{bonsai\_NODE}\ label\ subtrees) \Longrightarrow \\
\quad (\mathsf{ISL}\ label \wedge (\forall x.\mathsf{part\_fun\_domain}\ subtrees\ x = \mathsf{F})) \vee \\
\quad (\mathsf{ISR}\ label \wedge \mathsf{ISL}(\mathsf{OUTR}\ label) \wedge \\
\quad\ (\forall x.\ \mathsf{part\_fun\_domain}\ subtrees\ x = \mathsf{ISR}\ x \wedge \mathsf{ISL}(\mathsf{OUTR}\ x))) \vee \\
\quad (\mathsf{ISR}\ label \wedge \mathsf{ISR}(\mathsf{OUTR}\ label) \wedge \mathsf{ISL}(\mathsf{OUTR}(\mathsf{OUTR}\ label)) \wedge \\
\quad\ (\forall x.\ \mathsf{part\_fun\_domain}\ subtrees\ x = \mathsf{F})) \vee \\
\quad (\mathsf{ISR}\ label \wedge \mathsf{ISR}(\mathsf{OUTR}\ label) \wedge \mathsf{ISR}(\mathsf{OUTR}(\mathsf{OUTR}\ label)) \wedge \\
\quad\ (\forall x.\ \mathsf{part\_fun\_domain}\ subtrees\ x = \\
\qquad \mathsf{ISR}\ x \wedge \mathsf{ISR}(\mathsf{OUTR}\ x) \wedge \mathsf{ISR}(\mathsf{OUTR}(\mathsf{OUTR}\ x))))
\end{array}$$

Using `new_type_definition` with this predicate gets us the type that we need to solve the specification.

3.2 Making the Constructors

The next phase is making the constructors for the type. Each constructor needs to make a bonsai and then abstract it to the new type. It makes the bonsai using bonsai_NODE, and thus it needs to make a label and an indexed collection of subtrees. The label is simply the product of all the arguments to the constructor of pre-existing type (or one if there are none) injected into the corresponding summand of the label type. The indexed collection of subtrees is rather more complicated. The i'th constructor sends all summands of the branching type, except the i'th to undefined. If it has no argument types of the form $ty_{i,j} = ety_{i,j} \to rty$, then it sends everything of branching type to undefined. Otherwise, the i'th summand of the branching type is a sum of all $ety_{i,j}$ such that $ty_{i,j} = ety_{i,j} \to rty$. A summand of the i'th summand of the branching type of type $ety_{i,j}$ is sent to the bonsai representation of the corresponding rty value. For example, the second constructor for the type specification given above is

$$\begin{array}{l}
\mathsf{B} = \lambda(f_1 : \alpha \to \alpha\ \mathsf{toto})(f_2 : \mathsf{ind} \to \alpha\ \mathsf{toto}). \\
\quad \mathsf{toto\_ABS}(\ \mathsf{bonsai\_NODE} \\
\qquad\qquad (\mathsf{INR}(\mathsf{INL}\ \mathsf{one})) \\
\qquad\qquad (\lambda z.(\ \mathsf{ISR}\ z \wedge \mathsf{ISL}(\mathsf{OUTR}\ z)) \Rightarrow \\
\qquad\qquad\quad ((\mathsf{ISL}(\mathsf{OUTL}(\mathsf{OUTR}\ z))) \Rightarrow \\
\qquad\qquad\quad\ (\mathsf{toto\_REP}(f_1(\mathsf{OUTL}(\mathsf{OUTL}(\mathsf{OUTR}\ z))))) \\
\qquad\qquad\quad \mid (\mathsf{toto\_REP}(f_2(\mathsf{OUTR}(\mathsf{OUTL}(\mathsf{OUTR}\ z)))))) \\
\qquad\qquad\quad \mid \mathsf{undefined}))
\end{array}$$

Given these definitions for our constructors we show that no element is in the range of two distinct constructors and that every is element is in the range of one of the constructors.

3.3 Deriving Initiality

We prove initiality roughly in the following manner. Assume we have case functions for each of the cases of the specification (that is, build variables of the right types to be such case functions). Using these case functions we can build a case function over bonsai as follows. Given an indexed collection of recursive values *rec_fun*, a label *label*, and an indexed collection of subtrees *subtrees*, for each constructor, test whether there exist arguments such that bonsai_NODE *label subtrees* is the representation of the definition of the constructor applied to those arguments. If it is, return the appropriate case function applied to the appropriate arguments. If it fails to match any constructor, return $@x.\mathsf{T}$. For example, consider the specification

$$\mathsf{tutu}::=\mathsf{A}\ \mathsf{bool} \mid \mathsf{B}\ (\mathsf{num} \rightarrow \mathsf{tutu})$$

The case function we get then is

$$\begin{array}{l}
\lambda\ \mathit{rec\_fun}\ \mathit{label}\ \mathit{subtrees}.\\
\quad (\exists x.\mathsf{bonsai\_NODE}\ \mathit{label}\ \mathit{subtrees} =\\
\qquad \mathsf{bonsai\_NODE}\ (\mathsf{INL}\ x)\ (\lambda z.\mathsf{undefined})) \Rightarrow\\
\quad\ (A\_case(\mathsf{OUTL}\ \mathit{label}))\\
\quad \mid ((\exists g.\mathsf{bonsai\_NODE}\ \mathit{label}\ \mathit{subtrees} =\\
\qquad \mathsf{bonsai\_NODE}\ (\mathsf{INR}\ \mathsf{one})\ (\lambda z.\,(\mathsf{ISR}\ z) \Rightarrow\\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{tutu\_REP}(g(\mathsf{OUTR}\ z)))\\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{undefined})) \Rightarrow\\
\quad\ (B\_case\ (\lambda x.\mathit{rec\_fun}(\mathsf{INR}\ x))\ (\lambda x.\mathsf{tutu\_ABS}(\mathit{subtrees}(\mathsf{INR}\ x))))\\
\quad \mid @x.\mathsf{T})
\end{array}$$

If h is the function over bonsai given by the case function so generated, then $f = \lambda x.h(\mathit{rty}\_\mathsf{REP}\ x)$ is the function over the new recursive type given by the case functions over that type. We use the fact that the constructors have distinct images to demonstrate that the desired equations hold for f. We use the fact that every element is in the image of some constructor together with the uniqueness of h to show the uniqueness of the function satisfying the equations of the initiality theorem. The resultant initiality theorem for the tutu specification is

$$\begin{array}{l}
\forall A\_case\ B\_case.\ \exists! f.\,(\forall x.\ f(\mathsf{A}\ x) = \mathsf{A\_case}\ x) \wedge\\
\qquad\qquad\qquad\qquad\quad (\forall g.\ f(\mathsf{B}\ g) = \mathsf{B\_case}\ (f\ \mathsf{o}\ g)\ g)
\end{array}$$

4 Future Work

As was mentioned in Section 1, this work is a part of a project to support a class of mutually recursive specifications with nestings of type constructors. The general approach we take involves translating the specifications into progressively simpler forms, preserving information necessary to translate solutions back. First we eliminate the nested type constructors in favor of larger mutually recursive specifications having no such nestings. For an example of this, consider the specification we gave for bonsai:

$$\mathsf{bonsai} ::= \mathsf{bonsai\_NODE}\ \beta\ (\alpha \to \mathsf{bonsai\ lift})$$

The type bonsai being specified has an occurrence within the recursive type constructor lift. Using the specification for lift (which we can reconstruct from the initiality theorem for lift) we can convert this specification into the following form:

$$\begin{aligned} \mathsf{bonsai}' &::= \mathsf{bonsai\_NODE}'\ \beta\ (\alpha \to \mathsf{bonsai\_lift}) \\ \mathsf{bonsai\_lift} &::= \mathsf{lift}'\ \mathsf{bonsai} \mid \mathsf{undefined}' \end{aligned}$$

The type bonsai' is isomorphic to the type bonsai and the type bonsai_lift is isomorphic to the type bonsai lift.

The next translation transforms the mutually recursive specification into a simple recursive specification. This translation is basically the same as the one previously outlined by Thomas Melham in [3]. This translation does not yield an isomorphism, and so a second predicate must be defined on the type returned by the simple recursive specification culling out those terms that represent well-formed terms of the mutually recursive types. One way in which we differ slightly from the description given by Melham is that we must translate each occurrence of the recursive type *rty* as an argument to a constructor into an occurrence of $\mathsf{one} \to rty$. Perhaps one subtlety in this translation that was not discussed in [3] was determining whether the mutually recursive specification is well-founded, and the computation of witnesses for each type. It is possible for one or more of the types being defined in a mutually recursive specification to have no base case by itself, and yet for the system to be well-founded. This is the case, for example, with the translated specification for bonsai.

It should be noted, that although we haved talked in terms of making various definitions at various intermediate stages along the path to finding a solution to a specification, in reality we want to avoid actually introducing multiple intermediate types and constructors, and thus in our actual package we deal with the definitions of the objects instead of actually introducing these objects. It is rather important for purposes of both space and time efficiency not to introduce formally definitionally the intermediate stages.

Once the package for defining types from the specifications described in Section 1 is completed, there will still need to be a package developed for the flexible definition of functions over these types. The tools for proving results by induction over such types will also need to be extended.

5 Acknowledgements

It has come to my attention that David Shepherd of INMOS is carrying out a independent project parallel to the one discussed in this paper, and in particular, that he has developed a comparable theory of arbitrarily branching labeled trees as a foundation.

A Infinite Trees

A.1 An infinitely Tall bonsai (with no infinite branches)

Let us assume that we have a type bonsai described by

$$\mathsf{bonsai} ::= \mathsf{bonsai\_NODE}\ \beta\ (\alpha \to \mathsf{bonsai\ lift})$$

with the initiality theorem

$$\begin{aligned}&\forall Cases.\ \exists! f.\ \forall subtrees\ label.\ f(\mathsf{bonsai\_NODE}\ label\ subtrees) = \\ &\qquad Cases\ (\mathsf{lift\_compose}\ (\mathsf{lift\_fun}\ f)\ subtrees)\ label\ subtrees\end{aligned}$$

By primitive recursion on the natural numbers, we can define

$$\begin{aligned}&\mathsf{mk\_tree}\ 0 = \mathsf{bonsai\_NODE\ one}\ (\lambda x.\ \mathsf{undefined}) \wedge \\ &\forall n.\ \mathsf{mk\_tree}\ (\mathsf{SUC}\ n) = \mathsf{bonsai\_NODE\ one}\ (\lambda m.\mathsf{lift}(\mathsf{mk\_tree}\ n))\end{aligned}$$

Using these trees, we can then define

$$\mathsf{tall} = \mathsf{bonsai\_NODE\ one}\ (\mathsf{lift\_fun\ mk\_tree})$$

Given a partial function from α to num, we can define the least upper bound of the function by

$$\begin{aligned}&\forall p.\ \mathsf{lub}\ p = \\ &\quad (\forall x.\ p\ x = \mathsf{undefined}) \Rightarrow \mathsf{lift}\ 0 \\ &\quad \mid (\exists n.(\forall x.(p\ x \neq \mathsf{undefined}) \Longrightarrow (\mathsf{lower}(p\ x) \leq n)) \wedge (\exists x.\ p\ x = \mathsf{lift}\ n)) \Rightarrow \\ &\qquad \mathsf{lift}(@n.(\forall x.(p\ x \neq \mathsf{undefined}) \Longrightarrow (\mathsf{lower}(p\ x) \leq n)) \wedge (\exists x.\ p\ x = \mathsf{lift} n)) \\ &\qquad \mid \mathsf{undefined}\end{aligned}$$

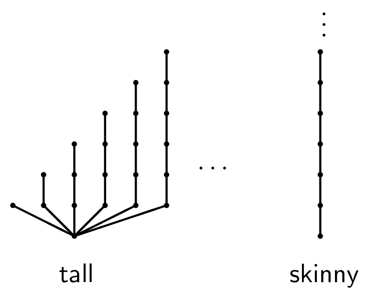
Now consider the following case function for trees:

$$\begin{aligned}&\mathsf{Cases}\ rec\_fun\ label\ subtrees = \\ &\quad (\exists x.\ rec\_fun\ x = \mathsf{lift}\ 13) \Rightarrow 13 \\ &\quad \mid (\mathsf{lub}\ rec\_fun = \mathsf{undefined}) \Rightarrow 13 \mid (2 + \mathsf{lower}(\mathsf{lub}\ rec\_fun))\end{aligned}$$

If we define our DoubleHeight function by

$$\begin{aligned}&\mathsf{DoubleHeight}\ (\mathsf{bonsai\_NODE}\ label\ subtrees) = \\ &\quad \mathsf{Cases}(\mathsf{lift\_compose}\ (\mathsf{lift\_fun}\ f)\ subtrees)\ label\ subtrees\end{aligned}$$

then we can show that DoubleHeight tall = 13.



A.2 $\mathsf{broad\_tree}$ is not Initial

Notice that the function $\lambda l\!:\!\mathsf{one\ list}.\ \mathsf{one}$ satisfies $\mathsf{Is\_labeling}$. Let

$$\mathsf{skinny} = \mathsf{broad\_tree\_ABS}(\lambda l\!:\!\mathsf{one\ list}.\ \mathsf{one})$$

Then we can show the following:

$$\mathsf{skinny} = \mathsf{broad\_tree\_NODE\ one}\ \lambda x\!:\!\mathsf{one}.\ \mathsf{skinny}$$

That is, every immediate subtree of skinny (there is only one) is equal to skinny. Consider the following two case functions over $(\mathsf{one}, \mathsf{one})\mathsf{broad\_tree}$:

$$\mathsf{U}\ \mathit{rec\_fun}\ \mathit{label}\ \mathit{subtrees} = ((\mathit{rec\_fun}\ \mathsf{one} = \mathsf{undefined}) \vee \mathsf{lower}(\mathit{rec\_fun}\ \mathsf{one}))$$

and

$$\mathsf{E}\ \mathit{rec\_fun}\ \mathit{label}\ \mathit{subtrees} = ((\mathit{rec\_fun}\ \mathsf{one} = \mathsf{undefined}) \vee \neg\mathsf{lower}(\mathit{rec\_fun}\ \mathsf{one}))$$

Then we have both that

$$\begin{aligned}&\forall \mathit{label}\ \mathit{subtrees}.(\lambda \mathit{tr}.\mathsf{T})(\mathit{Node}\ \mathit{label}\ \mathit{subtrees}) = \\ &\quad \mathsf{U}(\mathsf{lift\_compose}(\mathsf{lift\_fun}\ (\lambda \mathit{tr}.\mathsf{T}))\ \mathit{subtrees})\ \mathit{label}\ \mathit{subtrees}\end{aligned}$$

and

$$\begin{aligned}&\forall \mathit{label}\ \mathit{subtrees}.(\lambda \mathit{tr}.\mathit{tr} \neq \mathsf{skinny})(\mathit{Node}\ \mathit{label}\ \mathit{subtrees}) = \\ &\mathsf{U}(\mathsf{lift\_compose}(\mathsf{lift\_fun}\ (\lambda \mathit{tr}.\mathit{tr} \neq \mathsf{skinny}))\ \mathit{subtrees})\ \mathit{label}\ \mathit{subtrees}\end{aligned}$$

Therefore, there is no unique function defined by U. The situation is even worse with E. There we can prove

$$\begin{aligned}&\forall f.\ f(\mathsf{broad\_tree\_NODE\ one}\ \lambda x\!:\!\mathsf{one}.\ \mathsf{skinny}) \neq \\ &\quad \mathsf{E}(\mathsf{lift\_compose}(\mathsf{lift\_fun}\ f)\ \lambda x\!:\!\mathsf{one}.\ \mathsf{skinny})\ \mathsf{one}\ \lambda x\!:\!\mathsf{one}.\ \mathsf{skinny}\end{aligned}$$

References

1. M. J. C. Gordon. *The HOL System.* Cambridge Research Centre, SRI International, and DSTO Australia, 1989.
2. T. F. Melham. Automating recursive type definitions in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341 – 386. Springer-Verlag, 1989.
3. T.F. Melham. Email correspondence. `info-hol` email, 26 April 1992.

This article was processed using the LaTeX macro package with LLNCS style