T. Melham, 'Automating Recursive Type Definitions in Higher Order Logic',
in *Current Trends in Hardware Verification and Automated Theorem Proving*, edited by G. Birtwistle and P. A. Subrahmanyam (Springer-Verlag, 1989), pp. 341–386.

Automating Recursive Type Definitions in Higher Order Logic

Thomas F. Melham

University of Cambridge Computer Laboratory
New Museums Site, Pembroke Street
Cambridge, CB2 3QG, England.

Abstract: *The expressive power of higher order logic makes it possible to define a wide variety of types within the logic and to prove theorems that state the properties of these types concisely and abstractly. This paper contains a tutorial introduction to the logical basis for such type definitions. Examples are given of the formal definitions in logic of several simple types. A method is then described for systematically defining any instance of a certain class of commonly-used recursive types. The automation of this method in HOL, an interactive system for generating proofs in higher order logic, is also discussed.*

Introduction

Recursive structures, such as lists and trees, are widely used by computer scientists in formal reasoning about the properties of both hardware and software systems. The aim of this paper is to show how recursive structures of this kind can be defined in *higher order logic*, the logical formalism used by the HOL interactive proof-generating system [7].

Higher order logic is a typed logic; each variable in the logic has an associated logical *type* which specifies the kind of values it ranges over. Sets which contain recursive structures such as lists and trees can be represented in higher order logic by extending the syntax of types in the logic with new type expressions that denote these sets. In the version of higher order logic supported by the HOL system, this is done by first *defining* these new types in terms of already existing types and then deriving properties about the new types by formal proof. This guarantees that adding a new type to the logic will not introduce inconsistency. Sections 3 through 6 of this paper explain the formal mechanism for defining new types in higher order logic and give a series of detailed examples illustrating this mechanism.

In general, defining a new type in higher order logic can be tricky; the details of the definition have to be got just right to yield a type with the desired properties. But certain kinds of types can be defined systematically, and the process of defining them and proving that they have the required properties can therefore be automated. However, for this to be of *practical* value in a theorem prover such as HOL, it is essential that the automated tools for defining new types be reasonably efficient. To derive the properties of a defined type in HOL, all the logical inferences involved must be actually carried out in the system. To automate the definition of new types

in HOL, it is therefore desirable to reduce to a minimum the amount of inference that must be done. Section 7 of this paper shows how a certain class of widely-used concrete recursive types can be defined by a method which requires relatively little logical inference, and can therefore be efficiently automated in HOL.

All the theorems shown in this paper have been proved completely formally in the HOL system. And the method for automating recursive type definitions described in Section 7 has been fully implemented and is included in the latest release of HOL.

The Organization of the Paper

The organization of the paper is as follows. Section 1 contains an introduction to the version of higher order logic that is used in the paper. Section 2 describes how proofs in this formulation of higher order logic are mechanized in the HOL theorem prover. It is not possible to give more than a sketch of the HOL approach to theorem proving in this section; but a full description of HOL can be found in Gordon's paper [7]. In Section 3, a method is described by which new logical types can be defined as conservative extensions of higher order logic. Sections 4 through 6 consist of a series of examples which illustrate this method for defining new types. In Section 4, three simple logical types are defined: the 'trivial' type with only one value, the cartesian product type, and the disjoint sum type. In Section 5, two simple recursive types are defined: the type of natural numbers, and the type of lists. And in Section 6, the construction of two recursive types of trees is described. Finally, Section 7 outlines an efficient method for automating the definition of arbitrary concrete recursive types in higher order logic. This method uses types previously defined in Sections 4, 5, and 6. The implementation of the algorithm in the HOL system is also discussed in this section.

Note: Type constructions of the kind described in this paper are well-known in set theory (and logic), and no new theory of type constructions is presented here. The contribution of this paper consists rather in: (1) working out the details of defining these types in the particular logic implemented by the HOL theorem prover, and (2) building a logical basis for the *efficient* automation of recursive type definitions in HOL.

1 Introduction to Higher Order Logic

The version of higher order logic supported by the HOL system is based on Church's type theory [3], extended with the type discipline of the LCF logic PPλ [8]. This formulation of higher order logic was developed by Mike Gordon at the University of Cambridge, and is described in detail in [6]. This section gives a brief and informal introduction to the notation and some of the important features of this logic.

1.1 Notation

The syntax of higher order logic used in the HOL theorem prover includes terms corresponding to the conventional notation of predicate calculus. A term of the form $P\,x$ expresses the proposition that x has property P, and a term of the form $R(x, y)$ means that the relation R holds between x and y. The usual logical operators $\neg$, $\wedge$, $\vee$, $\supset$ and $\equiv$ denote negation, conjunction, disjunction, implication, and equivalence

respectively. The syntax of terms in HOL also includes the conventional notation for universal and existential quantifiers: $\forall x.\mathsf{P}\,x$ means that P holds for every value of x, and $\exists x.\mathsf{P}\,x$ means that P holds for at least one value of x. The additional quantifier $\exists!$ denotes unique existence: $\exists!x.\mathsf{P}\,x$ means that P holds for exactly one value of x. Nested quantifiers of the form $\forall v_1.\,\forall v_2.\,\cdots\,\forall v_n.\,tm$ can also be written $\forall v_1\,v_2\,\cdots\,v_n.\,tm$. Other notation includes $(c \Rightarrow t_1\,|\,t_2)$ to denote the conditional 'if c then t_1 else t_2', and $f\,\mathsf{o}\,g$ to denote the composition of the functions f and g. The constants T and F denote the truth values *true* and *false* respectively.

Higher order logic extends the notation of predicate calculus in three important ways: (1) variables are allowed to range over functions and predicates, (2) functions can take functions as arguments and yield functions as results, and (3) the notation of the λ-calculus can be used to write terms which denote functions.

The first two of these notational extensions are illustrated by the theorem of higher order logic shown below:

$$\vdash \forall xf.\,((\mathsf{rec}\ f)\ 0 = x) \wedge \forall n.\,(\mathsf{rec}\ f)\ (n{+}1) = f\ ((\mathsf{rec}\ f)\ n)$$

This theorem states that functions can be defined on the natural numbers such that they satisfy simple primitive recursive equations. It asserts that for any value x and any function f, the term $(\mathsf{rec}\ f)$ denotes a function that yields x when applied to 0 and satisfies the recursive equation $(\mathsf{rec}\ f)\ (n{+}1) = f\ ((\mathsf{rec}\ f)\ n)$ for all n. The universally quantified variable f in this theorem is an example of a higher-order variable: it ranges over functions. And the constant rec is an example of a higher order function: it both takes a function as an argument and yields a function as a result. Conventional practice is that function application in higher order logic associates to the left. So, for example, the term $(\mathsf{rec}\ f)\ n$ can also be written $\mathsf{rec}\ f\ n$.

The syntax of higher order logic also includes terms of the (typed) λ-calculus. If tm is a term and v is a variable, then the expression '$\lambda v.\,tm$' is also a term. It denotes the function whose value for an argument x is given by substituting x for v in tm. The term $\lambda n.\,n{+}1$, for example, denotes the successor function on natural numbers; and the term $(\lambda n.\,n{+}1)\,7$ can be simplified to $7{+}1$ by substituting 7 for n in $n{+}1$. Simplifications of this kind are called β-reductions.

1.2 Types in Higher Order Logic

Higher order logic is a *typed* logic; every syntactically well-formed term of the logic must have a type that is consistent with the types of its subterms. Informally, types can be thought of as denoting sets of values and terms as denoting elements of these sets.[1] As a syntactic device, types are necessary in higher order logic to eliminate certain paradoxes (e.g. Russell's paradox) which would otherwise arise because variables are allowed to range over functions and predicates.

Writing $tm{:}ty$ indicates explicitly that the term tm has type ty. Such explicit type information will usually be omitted, however, when it is clear from the form or context of the term what its type must be. The HOL mechanization of higher order logic uses Milner's elegant algorithm for type inference [11] to assign consistent types to logical terms entered by the user. The user of HOL therefore only occasionally has to give the types of terms explicitly.

[1] Because of the polymorphism introduced by type variables, the notion of types as sets is inadequate for a formal semantics of the logic. But it will do for the purposes of this paper.

1.2.1 The Syntax of Types

There are three syntactic classes of types in higher order logic: type constants, type variables, and compound types.

Type constants are identifiers that name sets of values. Examples are the two primitive types *bool* and *ind*, which denote the set of booleans and the set of 'individuals' (an infinite set) respectively. Another example is the type constant *num*, which denotes the set of natural numbers. The type *num* is not primitive but is defined in terms of *ind*; its definition is given in Section 5.1.

Type variables are used to stand for 'any type'; they are written α, β, γ, etc. Types that contain type variables are called *polymorphic* types. A *substitution instance* of a polymorphic type ty is a type obtained by substituting types for all occurrences of one or more of the type variables in ty. Theorems of higher order logic that contain polymorphic types are also true for any substitution instance of these types.

Compound types are expressions built from other types using *type operators*. They have the form: $(ty_1, ty_2, \ldots, ty_n)op$, where ty_1 through ty_n are types and op is the name of an n-ary type operator. An example is the binary type operator fun, which denotes the function space operation on types. The compound type $(ind, bool)fun$, for instance, is the type of all total functions from *ind* to *bool*. Types constructed using the type operator fun can also be written in a special infix form: $ty_1 \rightarrow ty_2$. The infix type operator $\rightarrow$ associates to the right; so the type $ind \rightarrow bool \rightarrow bool$, for example, is the same as $(ind, (bool, bool)fun)fun$.

In principle, every type needed for doing proofs in higher order logic can be written using type variables, the primitive type constants *bool* and *ind*, and the type operator fun. In practice, however, it is desirable to extend the syntax of types to help make theorems and proofs more concise and intelligible than would otherwise be possible. Section 3 shows how this can be done by adding new type constants and type operators to the logic using type 'definitions'.

1.3 Hilbert's ε-operator

An important primitive constant of higher order logic, which will be used frequently in this paper, is Hilbert's ε-operator. Its syntax and informal semantics are as follows. If $\mathsf{P}[x]$ is a boolean term involving a variable x of type ty then $\varepsilon x.\, \mathsf{P}[x]$ denotes some value, v say, of type ty such that $\mathsf{P}[v]$ is true. If there is no such value (i.e. $\mathsf{P}[v]$ is false for each value v of type ty) then $\varepsilon x.\, \mathsf{P}[x]$ denotes some fixed but arbitrarily chosen value of type ty. Thus, for example, '$\varepsilon n.\, 4{<}n \wedge n{<}6$' denotes the value 5, '$\varepsilon n.\, (\exists m.\, n{=}2{\times}m)$' denotes an unspecified even natural number, and '$\varepsilon n.\, n{<}n$' denotes an arbitrary natural number.

The informal semantics of Hilbert's ε-operator outlined above is formalized in higher order logic by the following theorem:

$$\vdash \forall P.\, (\exists x.\, P\ x) \supset P(\varepsilon x.\, P\ x)$$

It follows that if P is a predicate and $\vdash \exists x.\, \mathsf{P}x$ is a theorem of the logic, then so is $\vdash \mathsf{P}(\varepsilon x.\, \mathsf{P}x)$. The ε-operator can therefore be used to obtain a logical term which provably *denotes* a value with a given property P from a theorem merely stating that such a value *exists*. This property of ε is used extensively in the proofs outlined in this paper. For further discussion of the ε-operator, see [10].

An immediate consequence of the semantics of ε described above is that all logical types must denote non-empty sets. For any type ty, the term $\varepsilon x{:}ty.\mathsf{T}$ denotes an

element of the set denoted by ty. Thus the set denoted by ty must have at least one element. This will be important when the method for adding new types to the logic is discussed in Section 3.

2 The HOL Theorem Proving System

The HOL system [7] is a mechanized proof-assistant developed by Mike Gordon at the University of Cambridge for conducting proofs in the version of higher order logic described in the previous section. It has been primarily used to reason about the correctness of digital hardware. But much of what has been developed in HOL for hardware verification—the theory of arithmetic, for example—is also fundamental to many other applications. The underlying logic and basic facilities of the system are completely general and can in principle be used to support reasoning in any area that can be formalized in higher order logic.

HOL is based on the LCF approach to interactive theorem proving and has many features in common with the LCF theorem provers developed at Cambridge [12] and Edinburgh [8]. Like LCF, the HOL system supports secure theorem proving by representing its logic in the strongly-typed functional programming language ML [4]. Propositions and theorems of the logic are represented by ML abstract data types, and interaction with the theorem prover takes place by executing ML procedures that operate on values of these data types. Because HOL is built on top of a general-purpose programming language, the user can write arbitrarily complex programs to implement proof strategies. Furthermore, because of the way the logic is represented using ML abstract data types, such user-defined proof strategies are guaranteed to perform only valid logical inferences.

The HOL system has a special ML abstract data type `thm` whose values are theorems of higher order logic. There are no literals of type `thm`; that is, it is not possible to obtain an object of type `thm` by simply typing one in. There are, however, certain predefined ML identifiers which are given values of type `thm` when the system is built. These values correspond to the axioms of higher order logic. In addition, HOL makes available several predefined ML procedures that take theorems as arguments and return theorems as results. Each of these procedures corresponds to one of the primitive inference rules of the logic and returns only theorems that logically follow from its input theorems using the corresponding inference rule. Since ML is a strongly-typed language, the type checker ensures that values of type `thm` can be generated only by using these predefined functions. In HOL, therefore, every value of type `thm` must either be an axiom or have been obtained by computation using the predefined functions that represent the primitive inference rules of the logic. Thus every theorem in HOL must be generated from the axioms using the inference rules. In this way, the ML type checker guarantees the soundness of the HOL theorem prover.

In addition to the primitive inference rules, there are many *derived* inference rules available in HOL. These are ML procedures which perform commonly-used sequences of primitive inferences by applying the appropriate sequence of primitive inference rules. Derived inference rules relieve the HOL user of the need to explicitly give the all primitive inferences required in a proof. The ML code for a derived rule can be arbitrarily complex; but it will never return a theorem that does not follow by valid logical inference, since the type checker ensures that derived rules can only return theorems if they have been obtained by a series of calls to the primitive inference rules.

The approach to theorem proving described above ensures the soundness of the HOL theorem prover—but it is computationally expensive. Formal proofs of even simple theorems in higher order logic can take thousands of primitive inferences. And when these proofs are done in HOL, all the inferences must actually be carried out by executing the corresponding ML procedures.

There are, however, two important features of HOL which together allow *efficient* proof strategies to be programmed. The first of these is merely this: theorems proved in HOL can be saved on disk and therefore do not have to be generated each time they are needed in future proofs. The second feature is the expressive power of higher order logic itself, which allows useful and very general 'lemmas' to be stated in the logic. The amount of inference that a programmed proof rule must do can therefore be reduced by pre-proving general theorems from which the desired results follow by a relatively small amount of deduction. These theorems can then be saved and used by the derived inference rule in future proofs. This strategy of replacing 'run time' inference by pre-proved theorems is possible in HOL because type polymorphism and higher-order variables make the logic expressive enough to yield theorems of sufficient generality. This is illustrated in Section 7.4 of this paper, where a single general theorem is given from which the 'axiomatization' of any concrete recursive type can be efficiently deduced.

3 Defining New Logical Types

The primary function of types in higher order logic is to eliminate the potential for inconsistency that comes with allowing higher order variables. The type expressions needed to prevent inconsistency have a very simple and economical syntax; all that is needed are the types that can be constructed from type variables, the two primitive types *bool* and *ind*, and the type operator $\rightarrow$. In principle, every type needed for doing proofs in higher order logic can be written using only these primitive types. But in practice it is desirable to extend the syntax of types to include more kinds of types than are strictly necessary to prevent inconsistency.

Extending the syntax of type is of practical importance; it makes it possible to formulate propositions in logic in a more natural and concise way than can be done with only the primitive types. This pragmatic motivation for a rich syntax of types is similar to the motivation for the use of abstract data types in high-level programming languages; using higher level data types helps to control the size and complexity of proofs. This is essential in a theorem proving system (such as HOL) intended to be used as a practical tool for generating large formal proofs.

This section shows how new types can be consistently added to higher order logic by *defining* them in terms of already existing types. This is done in a way that allows theorems which 'axiomatize' these new types to be derived by formal proof from their definitions. The motivation for first defining a type and then deriving abstract 'axioms' for it is that this process guarantees consistency. Simply postulating axioms to describe the properties of new types may introduce inconsistency into the logic. But defining new types in terms of already existing types and then deriving axioms for them amounts to giving a consistency proof of these axioms.

3.1 Outline of the Method for Defining a New Type

The approach to defining new a logical type used in this paper involves the following three distinct steps:

1. finding an appropriate subset of an existing type to represent the new type;
2. extending the syntax of logical types to include a new type symbol, and using a *type definition axiom* to relate this new type to its representation; and
3. deriving from the type definition axiom and the properties of the representing type a set of theorems that serves as an 'axiomatization' of the new type.

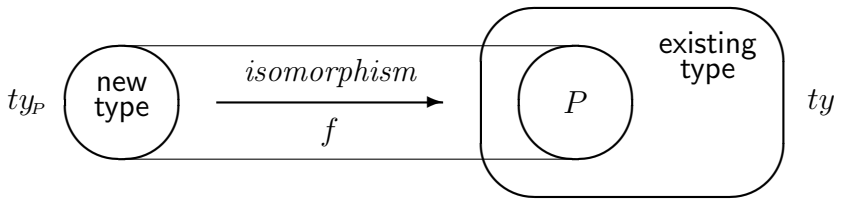
In the first of these steps, a model for the new type is given by specifying a set of values that will be used to represent it. This is done formally by defining a predicate P on an existing type such that the set of values satisfying P has exactly the properties that the new type is expected to have. In general, finding representations for new types and defining predicates that specify them can be difficult; but, as will be shown in Section 7.3, the representations of a certain class of recursive types can be constructed systematically.

In the second step, the syntax of types is extended to include a new type constant (or type operator) which denotes the set of values of the new type. This is done by adding a *type definition axiom* to the logic that serves to relate values of the new type to the corresponding values of the existing type that represent them. Type definition axioms are explained below in Section 3.2.

In the last step, a collection of theorems is proved that abstractly characterizes the new type. These theorems state the essential properties of the new type without reference to the way its values are represented and therefore act as an abstract 'axiomatization' of it. They are not, however, axioms in the sense that they are postulated without proof, but are *derived* by formal proof from the definition of the subset predicate given in step (1) and the type definition axiom postulated in step (2). This final step therefore amounts to giving a consistency proof of the axioms for the new type by showing that there is a model for them. Several examples of the derivation of axioms for new types are given in Sections 4–6; and, in Section 7, a method is described whereby the proof of the axioms for concrete recursive types can be efficiently automated.

3.2 Type Definition Axioms

The syntax of types in higher order logic can be extended to include new type constants as well as new type operators, by means of *type definition axioms*. This type definition mechanism is based on a suggestion by Mike Fourman which was formalized by Mike Gordon in [6]. The idea is that a new type is defined by adding an axiom to the logic which asserts that it is isomorphic to an appropriate 'subset' of an existing type:



Suppose, for example, that ty is a type of the logic and $P{:}ty{\rightarrow}bool$ is a predicate on values of type ty that defines some useful subset of the set denoted by ty. A type definition axiom defines a new type constant ty_P which denotes a set having exactly the same properties as the subset defined by P. This is done by extending the syntax of types to include the new type constant ty_P and then adding an *axiom* to the logic asserting that the set of values denoted by the new type is isomorphic to the set specified by P:

$$\vdash \exists f{:}ty_P{\rightarrow}ty.\ (\forall a_1\ a_2.f\ a_1 = f\ a_2 \supset a_1 = a_2) \wedge (\forall r.\ P\ r = (\exists a.\ r = f\ a)) \qquad (1)$$

This axiom states that there is a function f from the new type ty_P to the existing type ty which is one-to-one and onto the subset defined by P. The function f can be thought of as a representation function that maps a value of the new type ty_P to the value of type ty that represents it. Because f is an isomorphism, it can be shown that the set denoted by ty_P has the same properties as the subset of ty defined by P. By adding this axiom to the logic, the new type ty_P is therefore defined in terms of the existing type ty.

As was discussed in Section 1.3, the semantics of ε requires all types of the logic to denote non-empty sets. This means that the predicate P used in the type definition above must be true of at least one value of the representing type; i.e. it must be the case that $\vdash \exists x{:}ty.\ P\ x$. This existence theorem must be proved before the type definition axiom can be added to the logic. In the HOL theorem prover, the system requires the user to supply such an existence theorem before allowing a type definition axiom to be created.

If the subset defined by P is non-empty, then adding the type definition axiom (1) shown above is a *conservative extension* of the logic.[2] That is, for all boolean terms tm not containing the new type, $\vdash tm$ is a theorem of the extended logic if and only if it is a theorem of the original logic. In particular, $\vdash \mathsf{F}$ is a theorem of the extended logic if and only if it is a theorem of the original logic. Thus adding type definition axioms to the logic will not introduce inconsistency; adding type definition axioms is 'safe'.

In addition to type constants, new type operators can also be defined by adding axioms of the form shown above. For example, if $ty[\alpha, \beta]$ is an existing type that contains type variables α and β, and $P{:}ty[\alpha, \beta]{\rightarrow}bool$ is a predicate where $\vdash \exists x.\ P\ x$, then a new binary type operator $(\alpha, \beta)op$ can be defined by asserting the axiom:

$$\begin{aligned}\vdash \exists f{:}&(\alpha, \beta)op{\rightarrow}ty[\alpha, \beta].\\ &(\forall a_1\ a_2.f\ a_1 = f\ a_2 \supset a_1 = a_2) \wedge (\forall r.\ P\ r = (\exists a.\ r = f\ a))\end{aligned}$$

Detailed examples of type operator definitions are given in Sections 4.2 and 4.3 below, where definitions are described for the binary type operators *prod* (cartesian product) and *sum* (disjoint sum).

3.3 Defining Representation and Abstraction Functions

A type definition axiom of the form shown above merely asserts the *existence* of an isomorphism between a new type and the corresponding subset of an existing type. To formulate abstract axioms for a new type, it is convenient to have logical

[2]The term P in the type definition axiom must also satisfy certain syntactic conditions (which will not be discussed here) having to do with free variables and polymorphism.

constants which in fact *denote* such an isomorphism and its inverse. These mappings are used to define operations on the values of a new type in terms of operations on values of the representing type. These operations can then be used to formulate the abstract axioms for the new type. Using the primitive constant ε described in Section 1.3, constants denoting isomorphisms between new types and the subsets of existing types which represent them are easily defined as follows.

Given a type definition axiom stating the existence of an isomorphism between a new type ty_P and a subset of an existing type ty defined by a predicate P:

$$\vdash \exists f{:}ty_P{\to}ty.\,(\forall a_1\,a_2.f\ a_1 = f\ a_2 \supset a_1 = a_2) \wedge (\forall r.\,P\ r = (\exists a.\,r = f\ a))$$

a corresponding *representation* function $\mathsf{REP}{:}ty_P{\to}ty$ can be defined which maps a value of type ty_P to the value of type ty which represents it. Using the ε-operator, the function REP is defined by:

$$\vdash \mathsf{REP} = \varepsilon f.\,(\forall a_1\,a_2.f\ a_1 = f\ a_2 \supset a_1 = a_2) \wedge (\forall r.\,P\ r = (\exists a.\,r = f\ a)).$$

From the property of ε discussed in Section 1.3, it follows immediately that the function REP is one-to-one and onto the subset of ty given by P:

$$\vdash \forall a_1\,a_2.\,\mathsf{REP}\ a_1 = \mathsf{REP}\ a_2 \supset a_1 = a_2$$
$$\vdash \forall r.\,P\ r = (\exists a.\,r = \mathsf{REP}\ a)$$

Once the representation function REP is defined, the ε-operator can be used to define the inverse *abstraction* function $\mathsf{ABS}{:}ty{\to}ty_P$ as follows:

$$\vdash \forall r.\,\mathsf{ABS}\ r = (\varepsilon a.\,r = \mathsf{REP}\ a).$$

It is straightforward to prove that the abstraction function ABS is one-to-one for values of type ty satisfying P and that ABS is onto the new type ty_P:

$$\vdash \forall r_1\,r_2.\,P\ r_1 \supset (P\ r_2 \supset (\mathsf{ABS}\ r_1 = \mathsf{ABS}\ r_2 \supset r_1 = r_2))$$
$$\vdash \forall a.\,\exists r.\,(a = \mathsf{ABS}\ r) \wedge P\ r$$

It also follows from the definitions of the abstraction and representation functions that ABS is the left inverse of REP and, for values of type ty satisfying P, REP is the left inverse of ABS:

$$\vdash \forall a.\,\mathsf{ABS}(\mathsf{REP}\ a) = a$$
$$\vdash \forall r.\,P\ r = (\mathsf{REP}(\mathsf{ABS}\ r) = r)$$

Abstraction and representation functions of the kind illustrated by ABS and REP are used in every new type definition described in this paper. In each case, these functions are defined formally using the corresponding type definition axiom in the way shown above for ABS and REP. Theorems corresponding to those shown above for ABS and REP are used in the proofs of abstract axioms for each new type defined.

4 Three Simple Type Definitions

Three simple examples are given in this section to illustrate the method for defining new types described above in Section 3. In each example, a new type is defined using the three steps described in Section 3.1. First, an appropriate subset of an existing type is found to represent the values of the new type, and a predicate is defined to specify this subset. A type definition axiom for the new type is then postulated, and abstraction and representation functions are defined as described in Section 3.3. An abstract axiomatization is then formulated for the new type, which describes its properties without reference to the way it is represented and defined. This axiomatization follows by formal proof from the properties of the new type's representation. Some basic theorems about the new type are then derived from its abstract axiomatization.

The three types defined in this section will be used as basic 'building blocks' in the general method outlined in Section 7.3 for finding appropriate representations for arbitrary concrete recursive types.

4.1 The Type Constant *one*

This section describes the definition and axiomatization of the simplest (and the smallest) type possible in higher order logic: the type constant *one*, which denotes a set having exactly one element.

4.1.1 The Representation

To represent the type *one*, any singleton subset of an existing type will do. In the type definition given below, the subset of *bool* containing only the truth-value T will be used. This subset can be specified by the predicate $\lambda b{:}bool.\, b$, which denotes the identity function on *bool*. The set of booleans satisfying this predicate clearly has the property that the new type *one* is expected to have, namely the property of having exactly one element.

4.1.2 The Type Definition

As discussed in Section 3.2, a type definition axiom cannot be added to the logic unless the representing subset is non-empty. In the present case, the representing subset is specified by the predicate $\lambda b.\, b$. It is trivial to prove that this predicate specifies a non-empty set of booleans; the theorem $\vdash \exists x.\, (\lambda b.b)x$ follows immediately from $\vdash\ (\lambda b.b)\mathsf{T}$, which is itself equivalent to $\vdash\ \mathsf{T}$. Once it has been shown that $\lambda b.b$ specifies a non-empty set of booleans, the type constant *one* can be defined by postulating the type definition axiom shown below.

$$\vdash \exists f{:}one{\rightarrow}bool.\, (\forall a_1\, a_2.f\ a_1 = f\ a_2 \supset a_1 = a_2) \wedge (\forall r.\, (\lambda b.b)\ r = (\exists a.\, r = f\ a))$$

Using this type definition axiom, a representation function $\mathsf{REP\_one}{:}one{\rightarrow}bool$ can be defined to map the single value of type *one* to the boolean value T which represents it. As described in Section 3.3, this representation function can be defined such that it is one-to-one:

$$\vdash \forall a_1\, a_2.\, \mathsf{REP\_one}\ a_1 = \mathsf{REP\_one}\ a_2 \supset a_1 = a_2 \tag{2}$$

and onto the subset of *bool* defined by $\lambda b.b$:

$$\vdash \forall r.\,(\lambda b.b)\ r = (\exists a.\,r = \mathsf{REP\_one}\ a)$$

which, by the β-reduction $\vdash (\lambda b.b)\ r = r$, immediately yields the following theorem:

$$\vdash \forall r.\,r = (\exists a.\,r = \mathsf{REP\_one}\ a) \tag{3}$$

Theorems (2) and (3) about the representation function REP_one will be used in the proof given in the following section of the abstract axiomatization of *one*. The inverse abstraction function ABS_one:*bool*→*one* will not be needed in this proof.[3]

4.1.3 Deriving the Axiomatization of *one*

The axiomatization of the type *one* will consist of the following single theorem:

$$\vdash \forall f{:}\alpha{\to}one.\,\forall g{:}\alpha{\to}one.\,(f = g)$$

This theorem states that any two functions f and g mapping values of type α to values of type *one* are equal. From this it follows that there is only one value of type *one*, since if there were more than one such value it would be possible to define two different functions of type $\alpha{\to}one$. This theorem is therefore an abstract characterization of the type *one*; it expresses the essential properties of the type, but does so without reference to the way the type is represented.

The proof of the axiom for *one* uses the properties of REP_one given by theorems (2) and (3) above. Specializing the variable r in (3) to the term $\mathsf{REP\_one}(f\ x)$ yields:

$$\vdash \mathsf{REP\_one}(f\ x) = (\exists a.\,\mathsf{REP\_one}(f\ x) = \mathsf{REP\_one}\ a)$$

The right hand side of this equation is equal to T; this theorem can therefore be simplified to $\vdash \mathsf{REP\_one}(f\ x)$. Similar reasoning yields the theorem $\vdash \mathsf{REP\_one}(g\ x)$, from which it follows that:

$$\vdash \mathsf{REP\_one}(f\ x) = \mathsf{REP\_one}(g\ x)$$

From this theorem and theorem (2) stating that the function REP_one is one-to-one, it follows that $\vdash f\ x = g\ x$ and therefore that $\vdash \forall f\ g.\,(f = g)$, as desired.

4.1.4 A Theorem about *one*

Once the axiom for *one* has been proved, it is straightforward to prove a theorem which states explicitly that there is only one value of type *one*. This is done by defining a constant one to denote the single value of type *one*. Using the ε-operator, the definition of one can be written:

$$\vdash \mathsf{one} = \varepsilon x{:}one.\mathsf{T}$$

From the axiom for *one*, it follows that $\vdash \lambda x{:}\alpha.\,v = \lambda x{:}\alpha.\,\mathsf{one}$. Applying both sides of this equation to $x{:}\alpha$, and doing a β-reduction, gives $\vdash v = \mathsf{one}$. Generalizing v yields $\vdash \forall v{:}one.\,v = \mathsf{one}$, which states that every value v of type *one* is equal to the constant one, i.e. there is only one value of type *one*.

[3] In fact, the axiomatization of *one* can be derived directly from its type definition theorem; the constant REP_one is defined here merely to simplify the presentation of the proof that follows.

4.2 The Type Operator *prod*

In this section, a binary type operator *prod* is defined to denote the cartesian product operation on types. If ty_1 and ty_2 are types, then the type $(ty_1, ty_2)prod$ will be the type of ordered pairs whose first component is of type ty_1 and whose second component is of type ty_2.

4.2.1 The Representation

The type $(\alpha, \beta)prod$ can be represented by a subset of the polymorphic primitive type $\alpha \to \beta \to bool$. The idea is that an ordered pair $\langle a{:}\alpha,\ b{:}\beta \rangle$ will be represented by the function

$$\lambda x\, y.\, (x{=}a) \wedge (y{=}b)$$

which yields the truth-value T when applied to the two components a and b of the pair, and yields F when applied to any other two values of types α and β.

Every pair can be represented by a function of the form shown above; but not every function of type $\alpha \to \beta \to bool$ represents a pair. The functions that do represent pairs are those which satisfy the predicate $\mathsf{Is\_pair\_REP}$ defined by:

$$\vdash \mathsf{Is\_pair\_REP}\ f = \exists v_1\, v_2.\, f = \lambda x\, y.\, (x{=}v_1) \wedge (y{=}v_2),$$

i.e. those functions f which have the form $\lambda x\, y.\, (x{=}v_1) \wedge (y{=}v_2)$ for some pair of values v_1 and v_2. This will be the subset predicate for the representation of $(\alpha, \beta)prod$. As will be shown below, the set of functions satisfying $\mathsf{Is\_pair\_REP}$ has exactly the standard properties of the cartesian product of types α and β.

4.2.2 The Type Definition

To introduce a type definition axiom for *prod*, one must first show that the predicate $\mathsf{Is\_pair\_REP}$ defines a non-empty subset of $\alpha \to \beta \to bool$. This is easy, since it is the case that $\vdash \forall a\, b.\, \mathsf{Is\_pair\_REP}(\lambda x\, y.\, (x{=}a) \wedge (y{=}b))$ and therefore $\vdash \exists f.\, \mathsf{Is\_pair\_REP}f$. Once this theorem has been proved, a type definition axiom of the usual form can be introduced for the type operator *prod*:

$$\begin{array}{l}\vdash \exists f{:}(\alpha, \beta)prod \to (\alpha \to \beta \to bool).\\ \qquad (\forall a_1\, a_2.f\ a_1 = f\ a_2 \supset a_1 = a_2) \wedge (\forall r.\, \mathsf{Is\_pair\_REP}\ r = (\exists a.\, r = f\ a))\end{array}$$

This theorem defines the compound type $(\alpha, \beta)prod$ to be isomorphic to the subset of $\alpha \to \beta \to bool$ defined by $\mathsf{Is\_pair\_REP}$. Since the type variables α and β in this theorem can be instantiated to any two types, it has the effect of giving a representation not only for the particular type '$(\alpha, \beta)prod$', but also for the product of *any* two types. For example, instantiating both α and β to *bool* yields a type definition axiom for the cartesian product $(bool, bool)prod$. As will be shown below, the abstract axiomatization of *prod* derived from the type definition axiom given above is also formulated in terms of the compound type $(\alpha, \beta)prod$. It therefore also holds for any substitution instance of $(\alpha, \beta)prod$—i.e. for the product of any two types.

The abstract axiomatization of *prod* derived in the following section will use the abstraction and representation functions:

$$\mathsf{ABS\_pair}{:}(\alpha{\to}\beta{\to}\mathit{bool}){\to}(\alpha,\beta)\mathit{prod} \quad \text{and}$$
$$\mathsf{REP\_pair}{:}(\alpha,\beta)\mathit{prod}{\to}(\alpha{\to}\beta{\to}\mathit{bool})$$

which relate pairs to the functions of type $\alpha{\to}\beta{\to}\mathit{bool}$ which represent them. These representation and abstraction functions are defined formally as described above in Section 3.3. A set of theorems stating that Abs_pair and Rep_pair are isomorphisms can also be proved as outlined in Section 3.3. These theorems will be used in the proof of the axiom for *prod* given in the next section.

For notational convenience, an infix type operator '×' will be used in the remainder of this paper for the product of two types. Type expressions of the form $ty_1 \times ty_2$ will be simply syntactic abbreviations for $(ty_1, ty_2)\mathit{prod}$.

4.2.3 Deriving the Axiomatization of *prod*

To formulate the axiomatization of $(\alpha \times \beta)$, two constants will be defined:

$$\mathsf{Fst}{:}(\alpha \times \beta){\to}\alpha \quad \text{and} \quad \mathsf{Snd}{:}(\alpha \times \beta){\to}\beta.$$

These denote the usual *projection* functions on pairs; the function Fst extracts the first component of a pair, and the function Snd extracts the second component of a pair. The definitions of these functions are:

$$\vdash \mathsf{Fst}\ p\ = \varepsilon x.\, \exists y.\, (\mathsf{REP\_pair}\ p)\ x\ y$$
$$\vdash \mathsf{Snd}\ p = \varepsilon y.\, \exists x.\, (\mathsf{REP\_pair}\ p)\ x\ y$$

These definitions first use the representation function REP_pair to map a pair p to the function that represents it. They then 'select' the required component of the pair using the ε-operator. From the definitions of Fst and Snd, it is possible to show that

$$\begin{aligned} &\vdash \mathsf{Fst}(\mathsf{ABS\_pair}(\lambda x\, y.\, (x{=}a) \wedge (y{=}b)))\ = a \\ &\vdash \mathsf{Snd}(\mathsf{ABS\_pair}(\lambda x\, y.\, (x{=}a) \wedge (y{=}b))) = b \end{aligned} \tag{4}$$

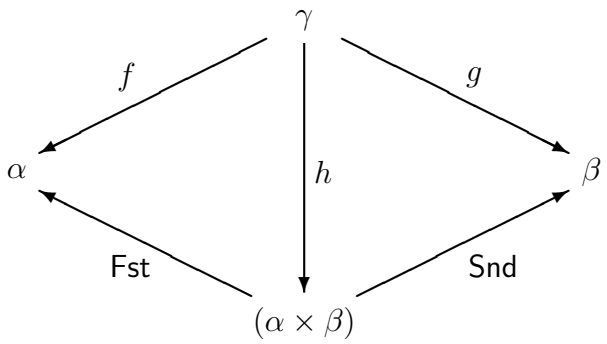
by using the fact that Rep_pair is the left inverse of ABS_pair for functions that satisfy the subset predicate Is_pair_REP. Once these two theorems have been proved, the axiomatization of the cartesian product of two types can be derived without further reference to the way Fst and Snd are defined.

Using the functions Fst and Snd, the axiomatization of the cartesian product of two types can be formulated based on the notion of a *product* in category theory. The following theorem will be the single axiom for the product of two types:

$$\vdash \forall f{:}\gamma{\to}\alpha.\ \forall g{:}\gamma{\to}\beta.\ \exists!\, h{:}\gamma{\to}(\alpha \times \beta).\ (\mathsf{Fst}\ \mathsf{o}\ h = f) \wedge (\mathsf{Snd}\ \mathsf{o}\ h = g)$$

This theorem states that for all functions f and g, there is a unique function h such

that the diagram



is commutative, i.e. $\forall x.\, \mathsf{Fst}(h\ x){=}f\ x$ and $\forall x.\, \mathsf{Snd}(h\ x){=}g\ x$. As noted above, this theorem is proved for the polymorphic type $(\alpha \times \beta)$. It therefore characterizes the product of any two types, since the type variables α and β in this theorem can be instantiated to any two types of the logic to yield an axiom for their product.

An outline of the proof of the axiom shown above is as follows. Given two functions $f{:}\gamma{\to}\alpha$ and $g{:}\gamma{\to}\beta$, define the function $h{:}\gamma{\to}(\alpha \times \beta)$ as follows:

$$h\ v = \mathsf{ABS\_pair}(\lambda x\, y.\, (x{=}f\ v) \wedge (y{=}g\ v))$$

Using the theorems (4) above, it follows that $\mathsf{Fst} \circ h = f$ and $\mathsf{Snd} \circ h = g$. To show that h is unique, suppose that there is also a function h' such that $\mathsf{Fst} \circ h' = f$ and $\mathsf{Snd} \circ h' = g$. Suppose v is some value of type γ. Since $\mathsf{ABS\_pair}$ is onto $(\alpha \times \beta)$, there exist a and b such that $h'\ v = \mathsf{ABS\_pair}(\lambda x\, y.\, (x{=}a) \wedge (y{=}b))$. Thus,

$$\begin{array}{lcccl} f\ v & = & \mathsf{Fst}(h'\ v) & = & \mathsf{Fst}(\mathsf{ABS\_pair}(\lambda x\, y.\, (x{=}a) \wedge (y{=}b))) \ = \ a \qquad \text{and} \\ g\ v & = & \mathsf{Snd}(h'\ v) & = & \mathsf{Snd}(\mathsf{ABS\_pair}(\lambda x\, y.\, (x{=}a) \wedge (y{=}b))) \ = \ b \end{array}$$

which means that

$$h'\ v \ = \ \mathsf{ABS\_pair}(\lambda x\, y.\, (x{=}f\ v) \wedge (y{=}g\ v)) \ = \ h\ v$$

and therefore that $h' = h$.

4.2.4 Theorems about *prod*

Using the axiom for products proved in the previous section, an infix operator $\otimes$ can be defined such that for all functions $f{:}\gamma{\to}\alpha$ and $g{:}\gamma{\to}\beta$ the expression $f \otimes g$ denotes the unique function of type $\gamma{\to}(\alpha{\times}\beta)$ which the axiom asserts to exist. This operator can be defined using the ε-operator as follows:

$$\vdash \forall f\, g.\, (f \otimes g) = \varepsilon h.\, (\mathsf{Fst} \circ h = f) \wedge (\mathsf{Snd} \circ h = g)$$

It follows from the axiom for products and the property of ε shown in Section 1.3 that $(f \otimes g)$ denotes a function which makes the diagram shown above commute:

$$\vdash \mathsf{Fst} \circ (f \otimes g) = f \quad \text{and} \quad \vdash \mathsf{Snd} \circ (f \otimes g) = g.$$

It can also be shown that for all f and g, the term $f \otimes g$ denotes the unique function with this property:

$$\vdash \forall f\, g\, h.\, (\mathsf{Fst} \circ h = f) \wedge (\mathsf{Snd} \circ h = g) \supset (h = (f \otimes g)).$$

Using the operator $\otimes$, an infix pairing function ',' can be defined to give the usual syntax for pairs, with (a, b) denoting the ordered pair having first component a and second component b. The definition is:

$$\vdash \forall a\, b.(a, b) = ((\mathsf{K}\ a) \otimes \mathsf{I})\ b \qquad \text{where } \mathsf{K} = \lambda a\, b.a \text{ and } \mathsf{I} = \lambda a.a.$$

The projection functions Fst and Snd and the constructor ',' defined above satisfy three theorems shown below, which are commonly used to characterize pairs.

$$\vdash \forall a\, b.\, \mathsf{Fst}(a, b) = a$$
$$\vdash \forall a\, b.\, \mathsf{Snd}(a, b) = b$$
$$\vdash \forall p.\, p = (\mathsf{Fst}\ p, \mathsf{Snd}\ p)$$

The first two of these theorems follow from the definition of the infix pairing operator ',' and the fact that $\vdash \mathsf{Fst} \circ ((\mathsf{K}\ a) \otimes \mathsf{I}) = \mathsf{K}\ a$ and $\vdash \mathsf{Snd} \circ ((\mathsf{K}\ a) \otimes \mathsf{I}) = \mathsf{I}$. The third theorem follows from the uniqueness of functions defined using $\otimes$.

4.3 The Type Operator *sum*

The final example in this section is the definition and axiomatization of a binary type operator *sum* to denote the disjoint sum operation on types. The set that will denoted by the compound type $(ty_1, ty_2)sum$ can be thought of as the union of two disjoint sets: a copy of the set denoted by ty_1, in which each element is labelled as coming from ty_1; and a copy of the set denoted by ty_2, in which each element is labelled as coming from ty_2. Thus each value of type $(ty_1, ty_2)sum$ will correspond either to a value of type ty_1 or to a value of type ty_2. Furthermore, each value of type ty_1 and each value of type ty_2 will correspond to a unique value of type $(ty_1, ty_2)sum$.

4.3.1 The Representation

One way of representing a value v of type $(\alpha, \beta)sum$ would be to use a triple (a, b, f) of type $\alpha \times \beta \times bool$, where f is a boolean 'flag' stating whether v corresponds to the value a of type α or the value b of type β. With this representation, each value a of type α would correspond to a triple $(a, \mathsf{d}_\beta, \mathsf{T})$ in the representation, where d_β is some fixed 'dummy' value of type β. Likewise, each value b of type β would have a corresponding triple $(\mathsf{d}_\alpha, b, \mathsf{F})$ in the representation, where d_α is a dummy value of type α. Using this representation, every value in the representing subset of $\alpha \times \beta \times bool$ would correspond either to a value of type α labelled by T or to a value of type β labelled by F.

The representation of values of type $(\alpha, \beta)sum$ can be both simplified and made independent of the product type operator by noting that a triple $(a, \mathsf{d}_\beta, \mathsf{T})$, for example, can itself be represented by the function:

$$\lambda x\, y\, fl.\, (x{=}a) \wedge (y{=}\mathsf{d}_\beta) \wedge (fl{=}\mathsf{T})$$

This function is true exactly when applied to the value a, the dummy value d_β and the truth-value T. Every function of this form corresponds to unique value of type α,

and every value of type α corresponds to a function of this form. But the same can be said of functions of the form:

$$\lambda x\, y\, fl.\, (x{=}a) \wedge (fl{=}\mathsf{T})$$

The dummy value d_β is therefore not necessary. A value of type $(\alpha, \beta)sum$ that corresponds to a value b of type β can likewise be represented by a function of the form:

$$\lambda x\, y\, fl.\, (y{=}b) \wedge (fl{=}\mathsf{F}).$$

The type $(\alpha, \beta)sum$ can therefore be represented by the subset of functions of type $\alpha{\to}\beta{\to}bool{\to}bool$ that satisfy the predicate $\mathsf{Is\_sum\_REP}$ defined by:

$$\begin{aligned}\vdash \mathsf{Is\_sum\_REP}\ f = {} & (\exists v_1.f = \lambda x\, y\, fl.\, (x{=}v_1) \wedge (fl{=}\mathsf{T})) \vee \\ & (\exists v_2.f = \lambda x\, y\, fl.\, (y{=}v_2) \wedge (fl{=}\mathsf{F}))\end{aligned}$$

The set of functions satisfying $\mathsf{Is\_sum\_REP}$ contains exactly one function for each value of type α and exactly one function for each value of type β. It therefore represents the disjoint sum of the set of values of type α and the set of values of type β.

4.3.2 The Type Definition

The type definition axiom for *sum* is introduced in exactly the same way as the defining axioms for *one* and *prod*. The first step is to prove a theorem stating that $\mathsf{Is\_sum\_REP}$ is true of at least one value in the representing set: $\vdash \exists f.\, \mathsf{Is\_sum\_REP}\ f$. A type definition axiom of the usual form can then be introduced:

$$\begin{aligned}\vdash \exists f{:}(\alpha, \beta)sum{\to}(\alpha{\to}\beta{\to}bool).\ & \\ (\forall a_1\, a_2.f\ a_1 = f\ a_2 \supset a_1 = a_2) \wedge (\forall r.\, \mathsf{Is\_sum\_REP}\ r = (\exists a.\, r = f\ a))&\end{aligned}$$

and the abstraction and representation functions

$$\mathsf{ABS\_sum}{:}(\alpha{\to}\beta{\to}bool{\to}bool){\to}(\alpha, \beta)sum \quad \text{and}$$
$$\mathsf{REP\_sum}{:}(\alpha, \beta)sum{\to}(\alpha{\to}\beta{\to}bool{\to}bool)$$

defined in the usual way. As outlined in Section 3.3, the definitions of $\mathsf{Abs\_sum}$ and $\mathsf{REP\_sum}$ and the type definition axiom for *sum* yield the usual isomorphism theorems about such abstraction and representation functions. These theorems will be used in the derivation of the abstract axiom for *sum*.

For notational clarity, an infix type operator '+' will now be used for the disjoint sum of two types. In what follows, the syntactic abbreviation $ty_1 + ty_2$ will be used instead of the form $(ty_1, ty_2)sum$.

4.3.3 Deriving the Axiomatization of *sum*

The axiomatization of $(\alpha + \beta)$ will use two constants:

$$\mathsf{Inl}{:}\alpha{\to}(\alpha + \beta) \quad \text{and} \quad \mathsf{Inr}{:}\beta{\to}(\alpha + \beta)$$

defined by:

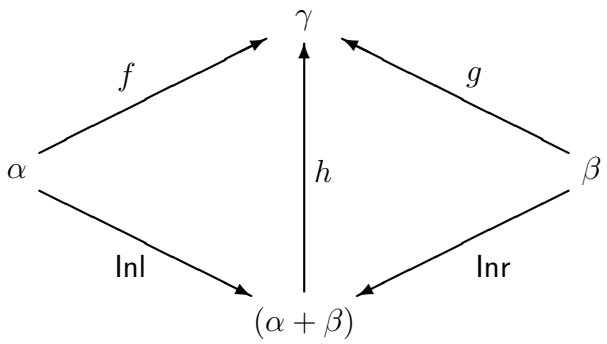
$$\vdash \mathsf{Inl}\ a = \mathsf{ABS\_sum}(\lambda x\, y\, fl.\, (x{=}a) \wedge (fl{=}\mathsf{T}))$$
$$\vdash \mathsf{Inr}\ b = \mathsf{ABS\_sum}(\lambda x\, y\, fl.\, (y{=}b) \wedge (fl{=}\mathsf{F}))$$

The constants Inl and Inr denote the left and right *injection* functions for sums. Every value of type $(\alpha+\beta)$ is either a left injection Inl a for some value $a{:}\alpha$ or a right injection Inr b for some value $b{:}\beta$.

The form of the axiom for $(\alpha+\beta)$ is based on the categorical notion of a *coproduct*. The axiom for $(\alpha+\beta)$ is:

$$\vdash \forall f{:}\alpha{\to}\gamma.\ \forall g{:}\beta{\to}\gamma.\ \exists!\, h{:}(\alpha+\beta){\to}\gamma.\ (h \circ \mathsf{Inl} = f) \wedge (h \circ \mathsf{Inr} = g)$$

This theorem asserts that for all functions f and g there is a unique function h such that the diagram shown below is commutative.



The proof of the axiom for sums is similar to the one outlined in the previous section for products. The proof will therefore not be given in full here. The existence of h follows simply by defining

$$h\ s = ((\exists v_1.\ x = \mathsf{Inl}\ v_1) \Rightarrow f(\varepsilon v_1.\ x = \mathsf{Inl}\ v_1)\,|\,g(\varepsilon v_2.\ x = \mathsf{Inr}\ v_2))$$

for given f and g. The uniqueness of h follows from the fact that Inl and Inr are one-to-one, and from the fact that ABS_sum is onto.

4.3.4 Theorems about *sum*

Using the axiom for sums, it is possible to define an operator $\oplus$ which is analogous to the operator $\otimes$ defined above for products. The definition of $\oplus$ is:

$$\vdash \forall f\ g.\ (f \oplus g) = \varepsilon h.\ (h \circ \mathsf{Inl} = f) \wedge (h \circ \mathsf{Inr} = g)$$

From the axiom for sums, it follows that for all functions f and g the term $(f \oplus g)$ denotes a function that makes the diagram for sums commute:

$$\vdash (f \oplus g) \circ \mathsf{Inl} = f \quad \text{and} \quad \vdash (f \oplus g) \circ \mathsf{Inr} = g$$

and that $(f \oplus g)$ denotes the unique function with this property:

$$\vdash \forall f\ g\ h.\ (h \circ \mathsf{Inl} = f) \wedge (h \circ \mathsf{Inr} = g) \supset (h = (f \oplus g)).$$

Using $\oplus$, it is possible to define two *discriminator* functions Isl:$(\alpha+\beta){\to}$*bool* and Isr:$(\alpha+\beta){\to}$*bool* as follows:

$$\vdash \mathsf{Isl} = (\mathsf{K}\ \mathsf{T}) \oplus (\mathsf{K}\ \mathsf{F}) \quad \text{and} \quad \vdash \mathsf{Isr} = (\mathsf{K}\ \mathsf{F}) \oplus (\mathsf{K}\ \mathsf{T})$$

From these definitions, and the properties of $\oplus$ shown above, it follows that every value of type $(\alpha + \beta)$ satisfies either Isl or Isr:

$$\vdash \forall s{:}(\alpha + \beta).\, \mathsf{Isl}\ s \vee \mathsf{Isr}\ s$$

and that Isl is true of left injections and Isr is true of right injections:

$$\vdash \forall a.\, \mathsf{Isl}(\mathsf{Inl}\ a) \qquad \vdash \forall b.\, \neg\mathsf{Isl}(\mathsf{Inr}\ b)$$
$$\vdash \forall b.\, \mathsf{Isr}(\mathsf{Inr}\ b) \qquad \vdash \forall a.\, \neg\mathsf{Isr}(\mathsf{Inl}\ a)$$

The operator $\oplus$ can also be used to define *projection* functions $\mathsf{Outl}{:}(\alpha+\beta){\to}\alpha$ and $\mathsf{Outr}{:}(\alpha + \beta){\to}\beta$ that map values of type $(\alpha + \beta)$ to the corresponding values of type α or β. Their definitions are:

$$\vdash \mathsf{Outl} = \mathsf{I} \oplus (\mathsf{K}\ \varepsilon b.\, \mathsf{F}) \quad \text{and} \quad \vdash \mathsf{Outr} = (\mathsf{K}\ \varepsilon a.\, \mathsf{F}) \oplus \mathsf{I}$$

where $\varepsilon a.\, \mathsf{F}$ and $\varepsilon b.\, \mathsf{F}$ denote 'arbitrary' values of type α and β respectively. From these definitions, it follows that the projection functions Outl and Outr have the properties:

$$\vdash \forall a.\, \mathsf{Outl}(\mathsf{Inl}\ a) = a \qquad \vdash \forall s.\, \mathsf{Isl}\ s \supset \mathsf{Inl}(\mathsf{Outl}\ s) = s$$
$$\vdash \forall a.\, \mathsf{Outr}(\mathsf{Inr}\ a) = a \qquad \vdash \forall s.\, \mathsf{Isr}\ s \supset \mathsf{Inr}(\mathsf{Outr}\ s) = s$$

5 Two Recursive Types: Numbers and Lists

This section outlines the definition of two recursive types: *num* (the type natural numbers) and $(\alpha)list$ (the polymorphic type of lists). Both *num* and $(\alpha)list$ are simple examples of the kind of recursive types which can be defined using the general method that will be described in Section 7. Their definitions are given here as examples to introduce the idea of defining recursive types in higher order logic. They also provide examples of the general form of abstract axiomatization that will be used in Section 7 for such types.

Both *num* and $(\alpha)list$ will be used in Section 6 to construct representations for two logical types of trees. Along with the basic building blocks: *one*, *prod* and *sum*, these types of trees will then be used in Section 7.3 to construct representations for arbitrary concrete recursive types.

5.1 The Natural Numbers

The construction of the natural numbers described in this section is based on the definition of the type *num* outlined by Gordon in [6]. The type *num* of natural numbers is defined using a subset of the primitive type *ind* of individuals. This primitive type is characterized by a single axiom, the 'axiom of infinity' shown below:

$$\vdash \exists f{:}ind{\to}ind.\, (\forall x_1\, x_2.\, (f\ x_1 = f\ x_2) \supset (x_1 = x_2)) \wedge \neg(\forall y.\, \exists x.\, y = fx) \qquad (5)$$

This theorem is one of the basic axioms of higher order logic. It asserts the existence of a function f from *ind* to *ind* which is one-to-one but not onto.

From this axiom, it follows that there are at least a countably infinite number of distinct values of type *ind*. Informally, this follows by observing that there is at least

one value of type *ind* which is not in the image of f. Call this value i_0. Now define i_1 to be $f(i_0)$. Since i_1 is in the image of the function f and i_0 is not, it follows that they are distinct values of type *ind*. Now, define i_2 to be $f(i_1)$. By the same argument as given above for i_1, it is clear that i_2 is not equal to i_0. Furthermore, i_2 is also not equal to i_1, since from the fact that f is one-to-one it follows that if $i_2 = i_1$ then $f(i_1) = f(i_0)$ and so $i_1 = i_0$. So i_2 is distinct from both i_1 and i_0. Defining i_3 to be $f(i_2)$, i_4 to be $f(i_3)$, etc. gives—by the same reasoning—an infinite sequence of distinct values of type *ind*. This infinite sequence can be used to represent the natural numbers.

5.1.1 The Representation and Type Definition

As was outlined informally above, it follows from the axiom of infinity (5) that there exists a function which can be used to 'generate' an infinite sequence of distinct values of type *ind*. The axiom of infinity merely asserts the existence of this function; the first step in representing the natural numbers is therefore to define a constant S:*ind*→*ind* which in fact *denotes* this function. Using the ε-operator, the definition of S is simply:

$$\vdash \mathsf{S} = \varepsilon f{:}ind{\to}ind.\, (\forall x_1\, x_2.\, (f\ x_1 = f\ x_2) \supset (x_1 = x_2)) \wedge \neg(\forall y.\, \exists x.\, y = f x)$$

Once S has been defined, a constant Z:*ind* can be defined which denotes a value not in the image of S. From this value Z, an infinite sequence of distinct individuals can then be generated by repeated application of S. The definition of Z simply uses the ε-operator to choose an arbitrary value not in the image of S:

$$\vdash \mathsf{Z} = \varepsilon y{:}ind.\, \forall x.\, \neg(y = \mathsf{S}\ x)$$

From the definitions of S and Z, the semantics of ε, and the axiom of infinity, it follows immediately that Z is not in the image of S and that S is one-to-one. Formally:

$$\begin{array}{l} \vdash \forall i.\, \neg(\mathsf{S}\ i = \mathsf{Z}) \\ \vdash \forall i_1\, i_2.\, (\mathsf{S}\ i_1 = \mathsf{S}\ i_2) \supset (i_1 = i_2) \end{array} \tag{6}$$

By the informal argument given in the introduction to this section, these two theorems imply that the individuals denoted by Z, $\mathsf{S}(\mathsf{Z})$, $\mathsf{S}(\mathsf{S}(\mathsf{Z}))$, $\mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z})))$, ... form an infinite sequence of distinct values, and can therefore be used to represent the type *num* of natural numbers. To make a type definition for *num*, a predicate N:*ind*→*bool* must be defined which is true of just those individuals in this infinite sequence. This can be done by defining N to be true of the values of type *ind* in the smallest subset of individuals which contains Z and is closed under S. The formal definition of N in higher order logic is:

$$\vdash \mathsf{N}\ i = \forall P{:}ind{\to}bool.\, P\ \mathsf{Z} \wedge (\forall x.\, P\ x \supset P(\mathsf{S}\ x)) \supset P\ i$$

This definition states that N is true of a value i:*ind* exactly when i is an element of *every* subset of *ind* which contains Z and is closed under S. This means that the subset of *ind* defined by N is the *smallest* such set and therefore contains just those individuals obtainable from Z by zero or more applications of S.

From the definition of N, it is easy to prove the following three theorems:

$$\begin{array}{l}\vdash \mathsf{N}\ \mathsf{Z}\\ \vdash \forall i.\, \mathsf{N}\ i \supset \mathsf{N}(\mathsf{S}\ i)\\ \vdash \forall P.\, (P\ \mathsf{Z} \wedge \forall i.\, (P\ i \supset P(\mathsf{S}\ i)) \supset \forall i.\, \mathsf{N}\ i \supset P\ i\end{array} \tag{7}$$

The first two of these theorems state that the subset of *ind* defined by N contains Z and is closed under the function S. The third theorem states that the subset of *ind* defined by N is the smallest such set. That is, any set of individuals containing Z and closed under S has the set of individuals specified by N as a subset.

Using the predicate N, the type constant *num* can be defined by introducing a type definition axiom of the usual form. From the theorem $\vdash \mathsf{N}\ \mathsf{Z}$, it follows immediately that $\vdash \exists i.\, \mathsf{N}\ i$. The following type definition axiom for the type *num* can therefore be introduced:

$$\vdash \exists f{:}num{\rightarrow}ind.(\forall a_1\, a_2.f\ a_1 = f\ a_2 \supset a_1 = a_2) \wedge (\forall r.\, \mathsf{N}\ r = (\exists a.\, r = f\ a))$$

and the usual abstraction and representation functions

$$\mathsf{ABS\_num}{:}ind{\rightarrow}num \quad \text{and} \quad \mathsf{REP\_num}{:}num{\rightarrow}ind$$

for mapping between values of type *num* and their representations of type *ind* can defined as described in Section 3.3.

5.1.2 Deriving the Axiomatization of *num*

The natural numbers are conventionally axiomatized by Peano's postulates. The five theorems labelled (6) and (7) in the previous section amount to a formulation of the Peano postulates for the natural numbers represented by individuals. It is therefore easy to derive Peano's postulates for the type *num* of natural numbers from these corresponding theorems about the subset of *ind* specified by N.

The first step in deriving the Peano postulates for *num* is to define the two constants:

$$0{:}num \quad \text{and} \quad \mathsf{Suc}{:}num{\rightarrow}num,$$

which denote the number zero and the successor function on natural numbers. Using the abstraction and representation functions $\mathsf{ABS\_num}$ and $\mathsf{REP\_num}$, the constants 0 and Suc can be defined as follows:

$$\begin{array}{l}\vdash 0 = \mathsf{ABS\_num}\ \mathsf{Z}\\ \vdash \mathsf{Suc}\ n = \mathsf{ABS\_num}(\mathsf{S}(\mathsf{REP\_num}\ n))\end{array}$$

From these definitions, the five theorems labelled (6) and (7), and the fact that the abstraction and representation functions $\mathsf{ABS\_num}$ and $\mathsf{REP\_num}$ are isomorphisms, it is easy to prove the abstract axiomatization of *num*, consisting of the three Peano postulates shown below:

$$\begin{array}{l}\vdash \forall n.\, \neg(\mathsf{Suc}\ n = 0)\\ \vdash \forall n_1\, n_2.\, \mathsf{Suc}\ n_1 = \mathsf{Suc}\ n_2 \supset n_1 = n_2\\ \vdash \forall P.\, (P\ 0 \wedge \forall n.\, P\ n \supset P(\mathsf{Suc}\ n)) \supset \forall n.\, P\ n\end{array}$$

The first of Peano's postulates shown above states that zero is not the successor of any natural number. This theorem follows immediately from the corresponding theorem $\vdash \forall i.\, \neg(\mathsf{S}\ i = \mathsf{Z})$ derived in the previous section for the representing values of type *ind*. Likewise, the second of Peano's postulates, which states that Suc is one-to-one, follows from the corresponding theorem about S. The third postulate states the validity of mathematical induction on natural numbers; it follows from the last of three theorems (7) derived in the previous section.

5.1.3 The Primitive Recursion Theorem

Once Peano's postulates have been proved, all the usual properties of the natural numbers can be derived from them. One important property is that functions can be uniquely defined on the natural numbers by primitive recursion. This is stated by the primitive recursion theorem, shown below:

$$\vdash \forall x f.\, \exists! fn.\, (fn\ 0 = x) \wedge \forall n.\, fn\ (\mathsf{Suc}\ n) = f\ (fn\ n)\ n \tag{8}$$

This theorem states that a function $fn{:}num{\rightarrow}\alpha$ can be *uniquely* defined by primitive recursion—i.e. by specifying a value for x to define the value of $fn(0)$ and an expression f to define the value of $fn(\mathsf{Suc}\ n)$ recursively in terms of $fn(n)$ and n. The proof of this theorem will not be given here, but an outline of the proof can be found in Gordon's paper [6]. The proof of a similar theorem for a logical type of *trees* is given in Section 6.1.3.

An important fact about the primitive recursion theorem is that it is equivalent to the three Peano postulates for *num* derived in Section 5.1.2. The single theorem (8) can therefore be used as the abstract axiomatization of the defined type *num*, instead of the three separate theorems expressing Peano's postulates. In Section 7.2, it will be shown how *any* concrete recursive type can be axiomatized in higher order logic by a similar 'primitive recursion' theorem.

Any function definition by primitive recursion on natural numbers can be justified formally in logic by appropriately specializing x and f in theorem (8). For example, specializing x and f to:

$$\lambda n.\, n \quad \text{and} \quad \lambda f x.\, \lambda m.\mathsf{Suc}(f\ m)$$

in a suitably type-instantiated version of the primitive recursion theorem yields (after some simplification) the theorem:

$$\vdash \exists! fn.\, (fn\ 0\ n = n) \wedge \forall n\, m.\, (fn\ (\mathsf{Suc}\ n)\ m = \mathsf{Suc}(fn\ n\ m))$$

which asserts the (unique) existence of an *addition* function on natural numbers. Primitive recursive definitions of other standard arithmetic operations (e.g. $+$, $\times$, and exponentiation) can also be formally justified using theorem (8).

5.2 Finite-length Lists

This section describes the definition of a recursive type $(\alpha)list$ of lists containing values of type α. In principle, it is possible to represent this type by a subset of some *primitive* compound type. But in practice, it is easier to use the defined type constant *num* and the type operator $\times$ (defined above in Section 4.2). The representation using *num* and $\times$ described below is based on Gordon's construction of lists in [6].

5.2.1 The Representation and Type Definition

Lists are simply finite sequences of values, all of the same type. A list with n values of type α will be represented by a pair (f, n), where f is a function of type $num{\rightarrow}\alpha$ and n is a value of type num. The idea is that the function f will give the sequence of values in the list; $f(0)$ will be the first value, $f(1)$ will be the second value, and so on. The second component of a pair (f, n) representing a list will be a number n giving the length of the list represented.

The set of values used to represent lists can not be simply the set of all pairs of type $(num{\rightarrow}\alpha) \times num$. The pairs used must be restricted so that each list has a *unique* representation. The one-element list [42], for example, will be represented by a pair $(f, 1)$, where $f(0)$=42. But there are an infinite number of different functions $f{:}num{\rightarrow}num$ that satisfy the equation $f(0)$=42. To make the representation of [42] unique, some 'standard' value must be chosen for the value of $f(m)$ when $m > 0$. The predicate $\mathsf{Is\_list\_REP}$ defined below uses the standard value $\varepsilon x{:}\alpha.\mathsf{T}$ to specify a set of pairs containing a unique representation for each list:

$$\vdash \mathsf{Is\_list\_REP}(f, n) = \forall m.\, m \geq n \supset (f\ m = \varepsilon x{:}\alpha.\mathsf{T})$$

If a pair (f, n) satisfies $\mathsf{Is\_list\_REP}$, then for $m < n$ the value of $f(m)$ will be the corresponding element of the list represented. For $m \geq n$, the value of $f(m)$ will be the standard value $\varepsilon x.\mathsf{T}$. With this representation, there is exactly one pair (f, n) for each finite-length list of values of type α.

It is easy to prove that $\vdash \exists f\, n.\, \mathsf{Is\_list\_REP}(f, n)$, since $\mathsf{Is\_list\_REP}$ holds of the pair $(\lambda n.\, \varepsilon x.\mathsf{T}, 0)$. A type definition axiom of the usual form can therefore be introduced for the type $(\alpha)list$:

$$\begin{aligned}\vdash\ & \exists f{:}(\alpha)list{\rightarrow}((num{\rightarrow}\alpha) \times num).\\ & \quad (\forall a_1\, a_2.f\ a_1 = f\ a_2 \supset a_1 = a_2) \wedge (\forall r.\, \mathsf{Is\_list\_REP}\ r = (\exists a.\, r = f\ a))\end{aligned}$$

and the abstraction and representation functions:

$$\begin{aligned}&\mathsf{ABS\_list}{:}((num{\rightarrow}\alpha) \times num){\rightarrow}(\alpha)list \quad \text{and}\\ &\mathsf{REP\_list}{:}(\alpha)list{\rightarrow}((num{\rightarrow}\alpha) \times num)\end{aligned}$$

can be defined based on the type definition axiom in the usual way.

5.2.2 Deriving the Axiomatization of $(\alpha)list$

The abstract axiomatization of lists will be based on two constructors:

$$\mathsf{Nil} : (\alpha)list \qquad \text{and} \qquad \mathsf{Cons} : \alpha{\rightarrow}(\alpha)list{\rightarrow}(\alpha)list.$$

The constant Nil denotes the empty list. The function Cons constructs lists in the usual way: if h is a value of type α and t is a list then $\mathsf{Cons}\ h\ t$ denotes the list with head h and tail t.

The definition of Nil is

$$\vdash \mathsf{Nil} = \mathsf{ABS\_list}((\lambda n{:}num.\, \varepsilon x{:}\alpha.\, \mathsf{T}), 0)$$

This equation simply defines Nil to be the list whose representation is the pair $(f, 0)$, where $f(n)$ has the value $\varepsilon x.\mathsf{T}$ for all n.

The constructor Cons can be defined by first defining a corresponding function Cons_REP which performs the Cons-operation on list representations. The definition is:

$$\vdash \mathsf{Cons\_REP}\ h\ (f, n) = ((\lambda m.(m{=}0 \Rightarrow h\ |f(m-1))),\ n+1)$$

The function Cons_REP takes a value h and pair (f, n) representing a list and yields the representation of the result of inserting h at the head of the represented list. This result is a pair whose first component is a function yielding value h when applied to 0 (the head of the resulting list) and the value given by $f(m{-}1)$ when applied to m for all $m{>}0$ (the tail of the resulting list). The second component is the length $n{+}1$, one greater than the length of the input list representation.

Once Cons_REP has been defined, it is easy to define Cons. The definition is:

$$\vdash \mathsf{Cons}\ h\ t = \mathsf{ABS\_list}(\mathsf{Cons\_REP}\ h\ (\mathsf{REP\_list}\ t))$$

The function Cons defined by this equation simply takes a value h and a list t, maps t to its representation, computes the representation of the desired result using Cons_REP, and then maps that result back to the corresponding abstract list.

Once Nil and Cons have been defined, the following abstract axiom for lists can be derived by formal proof:

$$\vdash \forall x\ f.\ \exists! fn.\ (fn(\mathsf{Nil}) = x) \wedge (\forall h\, t.\ fn(\mathsf{Cons}\ h\ t) = f\ (fn\ t)\ h\ t) \tag{9}$$

This axiom is analogous to the primitive recursion theorem for natural numbers, and is an example of the general form of the theorems which will be used in Section 7 to characterize all recursive types. Like the primitive recursion theorem, the abstract axiom for lists asserts that functions can be uniquely defined by primitive recursion. Once this theorem has been derived from the type definition axiom for lists and the definitions of Cons and Nil, all the usual properties of lists follow without further reference to the way lists are defined.

The axiom (9) for lists can be proved formally from the type definition for $(\alpha)list$. Full details will not be given here, but the proof is comparatively simple. The existence of the function fn in theorem (9) follows by demonstrating the existence of a corresponding function on list representations. This function can be defined by primitive recursion on the length component of the representation by using the primitive recursion theorem (8) for natural numbers. The uniqueness of the function fn in the abstract axiom for lists can then be proved by mathematical induction on the length component of list representations.

5.2.3 Theorems about $(\alpha)list$

Once the abstract axiom (9) for lists has been proved, the following three theorems can be derived from it:

$$\vdash \forall h\, t.\ \neg(\mathsf{Nil} = \mathsf{Cons}\ h\ t)$$

$$\vdash \forall h_1\, h_2\, t_1\, t_2.\ (\mathsf{Cons}\ h_1\ t_1 = \mathsf{Cons}\ h_2\ t_2) \supset ((h_1 = h_2) \wedge (t_1 = t_2))$$

$$\vdash \forall P.\ (P(\mathsf{Nil}) \wedge \forall t.\ P\ t \supset \forall h.\ P(\mathsf{Cons}\ h\ t)) \supset \forall l.\ P\ l$$

These three theorems are analogous to the Peano postulates for the natural numbers derived in Section 5.1.2. The first theorem states that Nil is not equal to any list constructed by Cons. The second theorem states that Cons is one-to-one. And the third theorem asserts the validity of structural induction on lists.

6 Two Recursive Types of Trees

This section describes the formal definitions of two different logical types which denote sets of trees. First, a type *tree* is defined which denotes the set of all trees whose nodes can branch any (finite) number of times. This type is then used to define a second logical type of trees, $(\alpha)Tree$, which denotes the set of *labelled* trees. These have the same sort of structure as values of type *tree*, but they also have a label of type α associated with each node.

The type $(\alpha)Tree$ defined in this section is of interest because each logical type in the class of recursive types discussed in Section 7 can be represented by some subset of it. Once the type of labelled trees has been defined, it can be used (along with the type *one* and the type operators $\times$ and $+$) to construct systematically a representation for any concrete recursive type. This avoids the problem of having to find an *ad hoc* representation for each recursive type, and so makes it possible to mechanize efficiently the formal definition of such types.

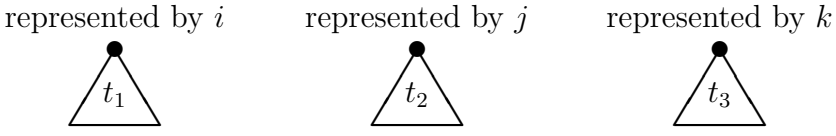
6.1 The Type of Trees: *tree*

Values of the logical type *tree* defined in this section will be finite trees whose internal nodes can branch any finite number of times. These trees will be *ordered*. That is, the relative order of each node's immediate subtrees will be important; and two similar trees which differ only in the order of their subtrees will be considered to be different trees.

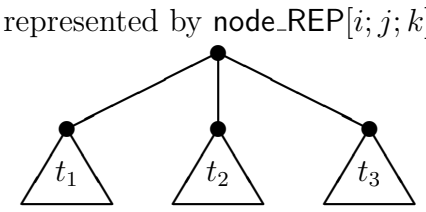
6.1.1 The Representation and Type Definition

Trees will be represented by coding them as natural numbers; each tree will be represented by a unique value of type *num*. The smallest possible tree consists of a single leaf node with no subtrees; it will be represented by the number 0. To represent a tree with one or more subtrees, a function node_REP:$(num)list \rightarrow num$ will be defined which computes the natural number representing such a tree from a list of the numbers which represent its subtrees. The function node_REP will take as an argument a list l of numbers. If each of the numbers in the list represents a tree, then node_REP l will represent the tree whose subtrees are represented by the numbers in l.

Consider, for example, a tree with three subtrees: t_1, t_2, and t_3. Suppose that the three subtrees t_1, t_2, and t_3 are represented by the natural numbers i, j, and k respectively:



The number representing the tree which has t_1, t_2, and t_3 as subtrees will then be denoted by node_REP$[i; j; k]$:



where the conventional list notation $[i; j; k]$ is a syntactic abbreviation for the list denoted by Cons i (Cons j (Cons k Nil)).

Since node_REP takes a *list* of numbers as arguments, it can be used to compute the code for a tree with any finite number of immediate subtrees. Thus, using node_REP, the natural number representing a tree of any shape can be computed recursively from the natural numbers representing its subtrees. The only property that node_REP must have for this to work is the property of being a one-to-one function on lists of numbers:

$$\vdash \forall l_1\, l_2.\, (\mathsf{node\_REP}\ l_1 = \mathsf{node\_REP}\ l_2) \supset (l_1 = l_2) \tag{10}$$

This theorem asserts that if node_REP computes the same natural number from two lists l_1 and l_2, then these lists must be equal and therefore must consist of the same finite sequence of numbers. If node_REP has this property, then it can be used to compute a *unique* numerical representation for every possible tree. It remains to define the function node_REP such that theorem (10) holds.

One way of formally defining node_REP is to use the well-known coding function $(n, m) \longmapsto (2n + 1) \times 2^m$ which codes a pair of natural numbers by a single natural number. Using this coding function, node_REP can be defined by recursion on lists such that the following two theorems hold:

$$\begin{aligned} &\vdash \mathsf{node\_REP\ Nil} &&= 0 \\ &\vdash \mathsf{node\_REP\ (Cons}\ n\ t) &&= ((2 \times n) + 1) \times (2\ \mathsf{Exp\ (node\_REP}\ t)) \end{aligned} \tag{11}$$

These two equations define the value of node_REP l by 'primitive recursion' on the list l. When l is the empty list Nil, the result is 0. When l is a non-empty list with head n and tail t, the result is computed by coding as a single natural number the pair consisting of n and the result of applying node_REP recursively to t. Primitive recursive definitions of this kind can be justified by formal proof using the abstract axiom (9) for lists derived in Section 5.2.2; the two theorems (11) can be derived from an appropriate instance of this axiom and a non-recursive definition of the constant node_REP.

Theorem (10) stating that node_REP is one-to-one can be derived from the two theorems (11) which define node_REP by primitive recursion. The proof is done by structural induction on the lists l_1 and l_2 using the theorem shown in Section 5.2.3 stating the validity of proofs by induction on lists.

The function node_REP can be used to compute a natural number to represent any finitely branching tree. To make a type definition for the type constant *tree*, a predicate on natural numbers Is_tree_REP:$num \rightarrow bool$ must be defined which is true of just those numbers representing trees. This predicate will be defined in the same way as the corresponding predicate was defined in Section 5.1.1 for the representation of numbers by individuals: Is_tree_REP n will be true if the number n is in the smallest set of natural numbers closed under node_REP.

The formal definition of Is_tree_REP uses the auxiliary function Every, defined recursively on lists as follows:

$$\begin{aligned} &\vdash \mathsf{Every}\ P\ \mathsf{Nil} &&= \mathsf{T} \\ &\vdash \mathsf{Every}\ P\ (\mathsf{Cons}\ h\ t) &&= (P\ h) \wedge \mathsf{Every}\ P\ t \end{aligned}$$

These two theorems define Every P l to mean that the predicate P holds of every element of the list l. Using Every, the predicate Is_tree_REP is defined as follows:

$$\vdash \mathsf{Is\_tree\_REP}\ n = \forall P.\, (\forall tl.\, \mathsf{Every}\ P\ tl \supset P(\mathsf{node\_REP}\ tl)) \supset P\ n$$

This definition states that a number n represents a tree exactly when it is an element of every subset of num which is closed under node_REP. It follows that the set of numbers for which ls_tree_REP is true is the smallest set closed under node_REP. This set contains just those natural numbers which can be computed using node_REP and therefore contains only those numbers which represent trees.

To use ls_tree_REP to define a new type, the theorem $\vdash \exists n.$ ls_tree_REP n must first be proved. This theorem follows immediately from the fact that ls_tree_REP is true of 0, i.e. the number denoted by node_REP Nil. Once this theorem has been proved, a type definition axiom of the usual form can be introduced:

$$\vdash \exists f{:}tree{\rightarrow}num.\ (\forall a_1\, a_2.f\ a_1 = f\ a_2 \supset a_1 = a_2) \wedge (\forall r.\ \mathsf{ls\_tree\_REP}\ r = (\exists a.\ r = f\ a))$$

along with the usual abstraction and representation functions:

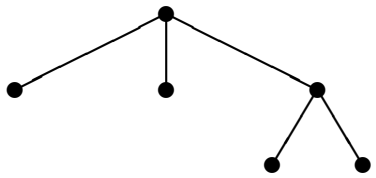
ABS_tree:$num{\rightarrow}tree$ and REP_tree:$tree{\rightarrow}num$.

6.1.2 The Axiomatization of *tree*

The abstract axiom for *tree* will be based on the constructor:

node:$(tree)list{\rightarrow}tree$

The function node builds trees from smaller trees. If $tl{:}(tree)list$ is a list of trees, then the term node tl denotes the tree whose immediate subtrees are the trees in the list tl. If tl is the empty list of trees, then node tl denotes the tree consisting of a single leaf node. Using node, it is possible to construct a tree of any shape. For example, the tree:



is denoted by the expression: node[node Nil; node Nil; node[node Nil; node Nil]].

An auxiliary function Map will be used in the formal definition of the constructor node. The function Map is the usual mapping function for lists; it takes a function $f{:}\alpha{\rightarrow}\beta$ and a list $l{:}(\alpha)list$ and yields the result of applying f to each member of l in turn. The recursive definition of Map is:

$$\vdash \mathsf{Map}\ f\ \mathsf{Nil} = \mathsf{Nil}$$
$$\vdash \mathsf{Map}\ f\ (\mathsf{Cons}\ h\ t) = \mathsf{Cons}\ (f\ h)\ (\mathsf{Map}\ f\ t)$$

Using Map and the function node_REP:$(num)list{\rightarrow}num$ defined in the previous section, the formal definition in logic of node is:

$$\vdash \mathsf{node}\ tl = (\mathsf{ABS\_tree}(\mathsf{node\_REP}(\mathsf{Map\ REP\_tree}\ tl)))$$

The constructor node defined by this equation takes a list of trees tl, applies node_REP to the corresponding list of numbers representing the trees in tl, and then maps the result to the corresponding abstract tree.

The following two important theorems follow from the formal definition of node given above; they are analogous to the Peano postulates for the natural numbers, and are used to prove the abstract axiom for the type *tree*:

$$\vdash \forall tl_1\ tl_2.\ (\mathsf{node}\ tl_1 = \mathsf{node}\ tl_2) \supset (tl_1 = tl_2)$$
$$\vdash \forall P.\ (\forall tl.\ \mathsf{Every}\ P\ tl \supset P\ (\mathsf{node}\ tl)) \supset \forall t.\ P\ t$$

The first of these theorems states that the constructor node is one-to-one. This follows directly from theorem (10), which states that the corresponding function node_REP is one-to-one. The second theorem shown above asserts the validity of induction on trees, and can be used to justify proving properties of trees by structural induction. This theorem can be proved from the definitions of node and Is_tree_REP and the fact that ABS_tree and REP_tree are isomorphisms relating trees and the numbers that represent them.

The abstract axiomatization of the defined type *tree* consists of the single theorem shown below:

$$\vdash \forall f.\ \exists! fn.\ \forall tl.\ fn(\mathsf{node}\ tl) = f\ (\mathsf{Map}\ fn\ tl)\ tl \tag{12}$$

This theorem is analogous to the primitive recursion theorem (8) for natural numbers and the abstract axiom (9) for lists. It asserts the unique existence of functions defined recursively on trees. The universally quantified variable f ranges over functions that map a list of values of type α and a list of trees to a value of type α. For any such function, there is a unique function $fn{:}tree{\rightarrow}\alpha$ that satisfies the equation $fn(\mathsf{node}\ tl) = f\ (\mathsf{Map}\ fn\ tl)\ tl$. For any tree $(\mathsf{node}\ tl)$, this equation defines the value of $fn(\mathsf{node}\ tl)$ recursively in terms of the result of applying fn to each of the immediate subtrees in the list tl.

6.1.3 An Outline of the Proof of the Axiom for *tree*

It is straightforward to prove the uniqueness part of the abstract axiom for trees; the uniqueness of the function fn in theorem (12) follows by structural induction on trees using the induction theorem for the defined type *tree*. The existence part of theorem (12) is considerably more difficult to prove. It follows from a slightly weaker theorem in which the list of subtrees tl is not an argument to the universally quantified function f:

$$\vdash \forall f.\ \exists fn.\ \forall tl.\ fn(\mathsf{node}\ tl) = f\ (\mathsf{Map}\ fn\ tl) \tag{13}$$

This weaker theorem can be proved by first defining a height function $\mathsf{Ht}{:}tree{\rightarrow}num$ on trees and then proving that, for any number n, there exists a function fun which satisfies the desired recursive equation for trees whose height is bounded by n:

$$\vdash \forall f\ n.\ \exists fun.\ \forall tl.\ (\mathsf{Ht}(\mathsf{node}\ tl) \leq n) \supset (fun(\mathsf{node}\ tl) = f\ (\mathsf{Map}\ fun\ tl)) \tag{14}$$

The main step in the proof of this theorem is an induction on the natural number n.

Theorem (14) can be used to define a higher order function fun which yields approximations of the function fn whose existence is asserted by theorem (13). For any n and f, the term $(\mathsf{fun}\ n\ f)$ denotes an approximation of fn which satisfies the recursive equation in theorem (13) for trees whose height is no greater than n. This is stated formally by the following theorem:

$$\vdash \forall f\ n\ tl.\ (\mathsf{Ht}(\mathsf{node}\ tl) \leq n) \supset (\mathsf{fun}\ n\ f\ (\mathsf{node}\ tl) = f\ (\mathsf{Map}\ (\mathsf{fun}\ n\ f)\ tl)) \tag{15}$$

The approximations of fn constructed by fun have the following important property: for any two numbers n and m, the corresponding functions constructed by fun behave the same for trees whose height is bounded by both n and m. This property follows by structural induction on trees, and is expressed formally by the theorem:

$$\vdash \forall t\ n\ m\ f.\,(\mathsf{Ht}\ t){<}n \wedge (\mathsf{Ht}\ t){<}m \supset (\mathsf{fun}\ n\ f\ t = \mathsf{fun}\ m\ f\ t) \tag{16}$$

Theorem (13) asserts the existence of a function fn for any given f; the higher order function fun can be used to explicitly construct this function fn from the given function f. For any f, the term $\lambda t.\,\mathsf{fun}\ (\mathsf{Ht}(\mathsf{node}\ [t]))\ f\ t$ denotes the function which satisfies the desired recursive equation. An outline of the proof of this is as follows. Specializing f, n, and tl in theorem (15) to f, $\mathsf{Ht}(\mathsf{node}[\mathsf{node}\ tl])$, and tl respectively yields the following implication:

$$\begin{array}{l}\vdash \mathsf{Ht}(\mathsf{node}\ tl) \leq \mathsf{Ht}(\mathsf{node}[\mathsf{node}\ tl]) \supset \\ \quad \mathsf{fun}\ (\mathsf{Ht}(\mathsf{node}[\mathsf{node}\ tl]))\ f\ (\mathsf{node}\ tl) = f(\mathsf{Map}\ (\mathsf{fun}\ (\mathsf{Ht}(\mathsf{node}[\mathsf{node}\ tl]))\ f)\ tl)\end{array}$$

The height function Ht has the property: $\vdash \forall t.\,\mathsf{Ht}\ t \leq \mathsf{Ht}(\mathsf{node}\ [t])$. The antecedent of the implication shown above is therefore always true, and the theorem can be simplified to:

$$\vdash \mathsf{fun}\ (\mathsf{Ht}(\mathsf{node}[\mathsf{node}\ tl]))\ f\ (\mathsf{node}\ tl) = f(\mathsf{Map}\ (\mathsf{fun}\ (\mathsf{Ht}(\mathsf{node}[\mathsf{node}\ tl]))\ f)\ tl)$$

The property of fun expressed by theorem (16) implies that the above theorem is equivalent to:

$$\vdash \mathsf{fun}\ (\mathsf{Ht}(\mathsf{node}[\mathsf{node}\ tl]))\ f\ (\mathsf{node}\ tl) = f(\mathsf{Map}\ (\lambda t.\,\mathsf{fun}\ (\mathsf{Ht}(\mathsf{node}[t]))\ f\ t)\ tl)$$

which is itself equivalent (by β-reduction) to:

$$\vdash (\lambda t.\,\mathsf{fun}\ (\mathsf{Ht}(\mathsf{node}[t]))f\ t)(\mathsf{node}\ tl) = f(\mathsf{Map}\ (\lambda t.\,\mathsf{fun}\ (\mathsf{Ht}(\mathsf{node}[t]))\ f\ t)\ tl)$$

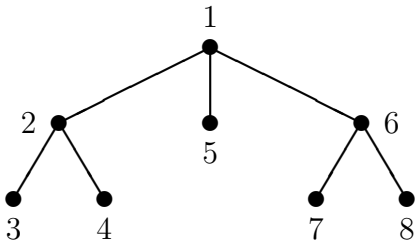
Theorem (13) follows immediately from this last result. The stronger theorem (12), which axiomatizes the defined type $tree$, then follows from theorem (13) by a relatively straightforward formal proof.

6.2 The Type of Labelled Trees: $(\alpha)Tree$

This section outlines the definition of the type $(\alpha)Tree$ which denotes the set of labelled trees. Labelled trees of the kind defined in this section have the same sort of general structure as values of the logical type $tree$ defined in the previous section. The only difference is that a tree of type $(\alpha)Tree$ has a value or 'label' of type α associated with each of its nodes. It is therefore comparatively simple to define the type $(\alpha)Tree$, since the values of the structurally similar type $tree$ can be used in its representation.

6.2.1 The Representation and Type Definition

The representation of a labelled tree of type $(\alpha)Tree$ will be a pair (t, l), where t is a value of type *tree* giving the shape of the tree being represented and l is a list of type $(\alpha)list$ containing the values associated with its nodes. The values in the list l will occur in the sequence which corresponds to a *preorder traversal* of the labelled tree being represented. Consider, for example, the labelled tree shown below:



This tree has a natural number associated with each node and can be represented by a pair (t, l) of type $tree \times (num)list$. The first component t of this pair will be the value of type *tree* whose structure corresponds to the above picture. The second component l will be a list of length eight containing the numbers associated with the nodes of the corresponding labelled tree. The numbers in this list will occur in the order $[1; 2; 3; 4; 5; 6; 7; 8]$, corresponding to a preorder traversal of the labelled tree being represented.

Any α-labelled tree can be similarly represented by a pair of type $tree \times (\alpha)list$; but not every such pair represents a tree. For a pair (t, l) to represent a labelled tree, the length of the list l must be the same as the number of nodes in the tree t. This can be expressed in logic by defining two functions:

$$\mathsf{Length}{:}(\alpha)list{\rightarrow}num \quad \text{and} \quad \mathsf{Size}{:}tree{\rightarrow}num$$

which compute the length of a list and the number of nodes in a tree, respectively. The function Length can be defined recursively by using the abstract axiom (9) for lists to derive the following two equations:

$$\begin{aligned} &\vdash \mathsf{Length\ Nil} &&= 0 \\ &\vdash \mathsf{Length}\ (\mathsf{Cons}\ h\ t) &&= (\mathsf{Length}\ t) + 1 \end{aligned}$$

The function Size can be defined by first defining a recursive function on lists $\mathsf{Sum}{:}(num)list{\rightarrow}num$ which computes the sum of a list of natural numbers:

$$\begin{aligned} &\vdash \mathsf{Sum\ Nil} &&= 0 \\ &\vdash \mathsf{Sum}\ (\mathsf{Cons}\ n\ l) &&= n + (\mathsf{Sum}\ l) \end{aligned}$$

and then using the abstract axiom (12) for the defined type *tree* to derive the following recursive definition of Size:

$$\vdash \mathsf{Size}(\mathsf{node}\ tl) = (\mathsf{Sum}(\mathsf{Map\ Size}\ tl)) + 1$$

Using the functions Length and Size, the values of type $tree \times (\alpha)list$ that represent labelled trees can be specified by the predicate Is_Tree_REP defined as follows:

$$\vdash \mathsf{Is\_Tree\_REP}(t, l) = (\mathsf{Length}\ l = \mathsf{Size}\ t)$$

This predicate is true of just those pairs (t, l) where the number of nodes in the tree t equals the length of the list l. It is therefore true of precisely those values of type $tree \times (\alpha)list$ which can be used to represent labelled trees.

For any value $v{:}\alpha$, the predicate Is_Tree_REP holds of the pair: (node Nil, $[v]$). From this, it immediately follows that $\vdash \exists p.\,$Is_Tree_REP p. The following type definition axiom can therefore be introduced to define $(\alpha)Tree$:

$$\vdash \exists f{:}(\alpha)Tree{\rightarrow}(tree \times (\alpha)list).\\ (\forall a_1\, a_2.f\ a_1 = f\ a_2 \supset a_1 = a_2) \wedge (\forall r.\,\textsf{Is\_Tree\_REP}\ r = (\exists a.\, r = f\ a))$$

The associated abstraction and representation functions:

$$\textsf{ABS\_Tree}{:}(tree \times (\alpha)list){\rightarrow}(\alpha)Tree \quad \text{and}$$
$$\textsf{REP\_Tree}{:}(\alpha)Tree{\rightarrow}(tree \times (\alpha)list)$$

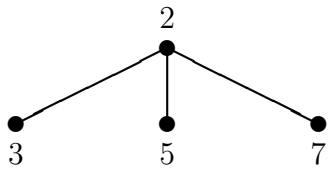
can then be defined in the usual way (as described in Section 3.3).

6.2.2 Deriving the Axiomatization of $(\alpha)Tree$

The abstract axiom for $(\alpha)Tree$ is based on the constructor

$$\textsf{Node}{:}\alpha{\rightarrow}((\alpha)Tree)list{\rightarrow}(\alpha)Tree$$

which is analogous to the constructor node for $tree$. If v is a value of type α, and l is a list of labelled trees, then the term (Node v l) denotes the labelled tree whose immediate subtrees are those occurring in l and whose root node is labelled by the value v. The function Node can be used to construct labelled trees of any shape. For example, the tree:



is denoted by the term: Node 2 [Node 3 Nil; Node 5 Nil; Node 7 Nil].

The formal definition of Node uses an auxiliary function Flat:$((\alpha)list)list{\rightarrow}(\alpha)list$ which takes a list of lists and yields the result of appending them all together into a single list. The recursive definition of Flat is:

$$\vdash \textsf{Flat Nil} = \textsf{Nil}$$
$$\vdash \textsf{Flat (Cons}\ h\ t) = \textsf{Append}\ h\ (\textsf{Flat}\ t)$$

where Append is defined (also recursively) by:

$$\vdash \textsf{Append Nil}\ l = l$$
$$\vdash \textsf{Append (Cons}\ h\ l_1)\ l_2 = \textsf{Cons}\ h\ (\textsf{Append}\ l_1\ l_2)$$

Using Flat, and the mapping function Map defined above in Section 6.1.2, the formal definition of the constructor Node is given by the following theorem:

$$\vdash \textsf{Node}\ v\ l = \textsf{ABS\_Tree}((\textsf{node}(\textsf{Map (Fst o REP\_Tree)}\ l)),\\ ((\textsf{Cons}\ v\ (\textsf{Flat}(\textsf{Map (Snd o REP\_Tree)}\ l)))))$$

This definition uses REP_Tree to obtain the representation of each labelled tree in the list l. This yields a list of pairs representing labelled trees. The function node is then used to construct a new tree whose subtrees are the tree components in this list of pairs, and Flat is used to construct the corresponding list of node-values. The result is then mapped back to an abstract labelled tree using the abstraction function ABS_Tree.

Using the constructor Node v l defined above, the abstract axiom for $(\alpha)Tree$ can be written:

$$\vdash \forall f.\, \exists! fn.\, \forall v\, tl.\, fn(\mathsf{Node}\ v\ tl) = f\ (\mathsf{Map}\ fn\ tl)\ v\ tl \tag{17}$$

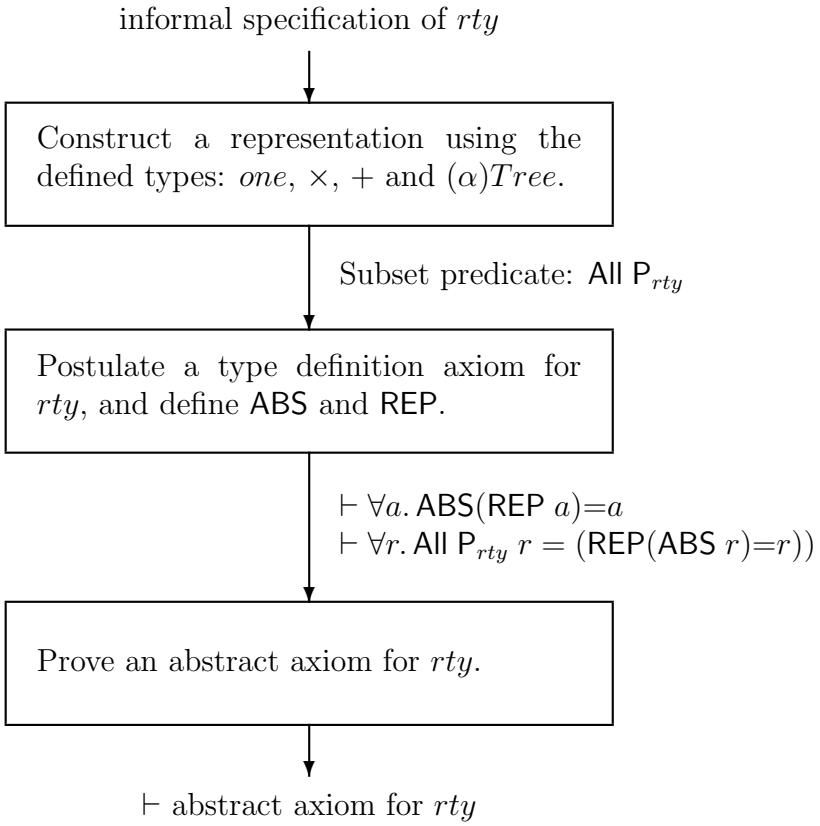
This theorem is of the same general form as theorem (12), the abstract axiom for the defined type *tree*. It states the uniqueness of functions defined by 'primitive recursion' on labelled trees. The proof of this theorem is straightforward, but it requires some tricky (and uninteresting) lemmas involving the partitioning of lists. Details of the proof will therefore not be given here. The general strategy of the proof is to use the abstract axiom for values of type *tree* to define a recursive function on *representations* which 'implements' the function fn asserted to exist by the axiom (17).

7 Automating Recursive Type Definitions

This section outlines a method for formally defining any simple concrete recursive type in higher order logic. This method has been used to implement an efficient derived inference rule in HOL which defines such recursive types automatically. The input to this derived rule is a user-supplied informal[4] specification of the recursive type to be defined. This type specification is written in a notation which resembles a data type declaration in functional programming languages like Standard ML [9]. It simply states the names of the new type's constructors and the logical types of their arguments. The output is a theorem of higher order logic which abstractly characterizes the properties of the desired recursive type—i.e. a derived 'abstract axiomatization' of the type.

An overview of the algorithm used by this programmed inference rule to define a new recursive type is shown in the diagram below. The algorithm follows the three steps for defining a new logical type described in Section 3.1.

[4]In this context, *informal* means not in the language of higher order logic.



In the first step, an appropriate representation is found for the values of the recursive type rty to be defined. This representation is always some subset of a substitution instance of $(\alpha)Tree$—i.e. a subset of some type $(ty)Tree$ of general trees labelled by values of type ty. The type ty of labels for these trees is built up systematically using the type constant *one* and the type operators $\times$ and $+$. The output of this stage is a 'subset predicate' which defines the set of labelled trees used to represent values of the new type rty. This predicate has the standard form: '$\mathsf{All}\ \mathsf{P}_{rty}$', where P_{rty} is a predicate whose exact form is determined by the specification of the type to be defined. (The meaning of 'All' is explained below in Section 7.3.2.) No logical inference needs to be done in this step; so the ML code which implements it in the HOL system is quite fast.

In the second step, a type definition axiom is introduced for the new type, based on the subset predicate $\mathsf{All}\ \mathsf{P}_{rty}$. The associated abstraction and representation functions ABS and REP are then defined and proved to be isomorphisms between the new type rty and the set of values specified by $\mathsf{All}\ \mathsf{P}_{rty}$. The output of this stage consists of the two theorems about ABS and REP shown in the diagram above. The proofs done in this step are easy and routine (see Section 3.3), and their mechanization in HOL is therefore efficient and straightforward.

In the final step, an abstract axiom for the new type rty is derived by formal proof from the definition of the subset predicate $\mathsf{All}\ \mathsf{P}_{rty}$ and the two theorems about ABS and REP proved in the previous stage. This is the only step in the algorithm where a non-trivial amount of logical inference has to be done. The ML implementation of this step therefore uses the 'optimization' strategy for HOL derived inference rules discussed in Section 2: a pre-proved general theorem about recursive types is used to reduce to a minimum the amount of inference that has to be done at 'run time' to derive the desired result. This pre-proved theorem has the form shown below:

$$\vdash \forall P. \cdots \langle \beta \text{ is isomorphic to '}\mathsf{All}\ P\text{'}\rangle \supset \langle \text{abstract axiom for } \beta\rangle$$

Informally, this theorem states that *any* type β which is represented by a set of labelled trees '$\mathsf{All}\ P$' satisfies an abstract axiomatization of the required form. By specializing P in this theorem to the predicate $\mathsf{P}_r ty$ constructed in the first step, the abstract axiom for rty follows simply by modus ponens (using the theorems about ABS and REP derived in the second step) and a relatively small amount of straightforward simplification.

A detailed description of the HOL implementation of this algorithm for defining recursive types is beyond the scope of this paper; but the sections which follow give an overview of the logical basis of this implementation. In Section 7.1, the syntax of informal type specifications is described, and some simple examples are given of type specifications written in this notation. Section 7.2 then describes the general form of the abstract axioms that are derived by the system. Section 7.3 explains how appropriate representations for these types can be systematically constructed from their informal type specifications. Finally, Section 7.4 gives the general theorem stating that any recursive type represented in the way described in Section 7.3 satisfies an abstract axiom of the form shown in Section 7.2. An example of the application of this theorem is also given.

7.1 Informal Type Specifications[5]

Every logical type which can be defined by the method outlined in the following sections can be described informally by a *type specification* of the following general form:

$$(\alpha_1, \ldots, \alpha_n)rty \quad ::= \quad \mathsf{C}_1\ ty_1^1\ \ldots\ ty_1^{k_1} \quad | \quad \cdots \quad | \quad \mathsf{C}_m\ ty_m^1\ \ldots\ ty_m^{k_m} \qquad (18)$$

where each ty_i^j is either an existing logical type (not containing rty) or is the type expression $(\alpha_1, \ldots, \alpha_n)rty$ itself. This equation specifies a type $(\alpha_1, \ldots, \alpha_n)rty$ with n type variables $\alpha_1, \ldots, \alpha_n$ where $n \geq 0$. If n=0 then rty is a type constant; otherwise rty is an n-ary type operator. The type specified has m distinct constructors $\mathsf{C}_1, \ldots, \mathsf{C}_m$ where $m \geq 1$. Each constructor C_i takes k_i arguments, where $k_i \geq 0$; and the types of these arguments are given by the type expressions ty_i^j for $1 \leq j \leq k_i$. If one or more of the type expressions ty_i^j is the type $(\alpha_1, \ldots, \alpha_n)rty$ itself, then the equation specifies a *recursive* type. In any specification of a recursive type, at least one constructor must be non-recursive—i.e. all its arguments must have types which already exist in the logic.

The logical type specified by equation (18) denotes the set of all values which can be finitely generated using the constructors $\mathsf{C}_1, \ldots, \mathsf{C}_m$, where each constructor is one-to-one and any two different constructors yield different values. I.e. the type specified by (18) is the *initial algebra* [2] with constructors $\mathsf{C}_1, \ldots, \mathsf{C}_m$. Every value of this logical type is denoted by some term of the form:

$$\mathsf{C}_i\ x_i^1\ \ldots\ x_i^{k_i}$$

where x_i^j is a term of logical type ty_i^j for $1 \leq j \leq k_i$. In addition, any two terms:

$$\mathsf{C}_i\ x_i^1\ \ldots\ x_i^{k_i} \qquad \text{and} \qquad \mathsf{C}_j\ x_j^1\ \ldots\ x_j^{k_j}$$

denote equal values exactly when their constructors are the same (i.e. $i = j$) and these constructors are applied to equal arguments (i.e. $x_i^n = x_j^n$ for $1 \leq n \leq k_i$).

[5]Some of the notation used in this section is adapted from Bird and Wadler's clear description of the syntax of type definitions in their excellent book [1] on functional programming.

7.1.1 Some Examples of Type Specifications

The two simple recursive types *num* and $(\alpha)list$ which were defined in Section 5 are both examples of types that can be described by type specifications of the general form illustrated by (18) above.

The specification of the type *num* of natural numbers is the simple equation shown below:

$$num \quad ::= \quad 0 \quad | \quad \mathsf{Suc}\ num$$

This equation specifies the type constant *num* to have two constructors: 0:*num* and Suc:*num*→*num*. The type *num* which is described by this type specification denotes the set of values generated from the constant 0 by zero or more applications of the constructor Suc—i.e. the set of values denoted by terms of the form: 0, Suc(0), Suc(Suc(0)), ... etc.

The type specification for the type $(\alpha)list$ of finite lists is similar to the one given above for *num*. It is:

$$(\alpha)list \quad ::= \quad \mathsf{Nil} \quad | \quad \mathsf{Cons}\ \alpha\ (\alpha)list$$

This equation states that the type $(\alpha)list$ denotes the set of all values generated by the two constructors: Nil:$(\alpha)list$ and Cons:$\alpha \rightarrow (\alpha)list \rightarrow (\alpha)list$.

A slightly more complex example is the recursive type *btree*, described by the type specification shown below:

$$btree \quad ::= \quad \mathsf{Leaf}\ num \quad | \quad \mathsf{Tree}\ btree\ btree$$

This equation specifies a type of *binary trees* whose leaf nodes (but not internal nodes) are labelled by natural numbers. When defined formally in higher order logic, this type has two constructors: Leaf:*num*→*btree* and Tree:*btree*→*btree*→*btree*. The function Leaf constructs leaf nodes; if n is a value of type *num*, then (Leaf n) denotes a leaf node labelled by n. The constructor Tree builds binary trees from smaller binary trees; if t_1 and t_2 are binary trees then (Tree t_1 t_2) denotes the binary tree with left subtree t_1 and right subtree t_2.

In addition to recursive types, simple enumerated and 'record' types can also be specified by equations of the form given by (18). For example, the type constant *one* and the two type operators *prod* and *sum*, whose formal definitions were given in Section 4, can be informally specified by the three equations shown below:

$$\begin{array}{rcl} one & ::= & \mathsf{one} \\ (\alpha,\beta)prod & ::= & \mathsf{pair}\ \alpha\ \beta \\ (\alpha,\beta)sum & ::= & \mathsf{Inl}\ \alpha \quad | \quad \mathsf{Inr}\ \beta \end{array}$$

The first of these specifications simply states that *one* is the enumerated type with exactly one value: the value denoted by the constant one. The second specification states that every value of type $(\alpha,\beta)prod$ is denoted by some term of the form (pair a b), i.e. an ordered pair with first component a:α and second component b:β. The third equation states that every value of type $(\alpha,\beta)sum$ is either a left injection constructed by Inl or a right injection constructed by Inr.

Many more examples of types—both recursive and non-recursive—which can be specified by equations of the form given by (18) can be found in books on functional programming. (See, for example, chapter 8 of [1].)

7.2 Formulating Abstract Axioms for Recursive Types

The input to the HOL programmed inference rule which defines types is, in general, an informal specification of the form:

$$(\alpha_1,\ldots,\alpha_n)rty \quad ::= \quad \mathsf{C}_1\ ty_1^1\ \ldots\ ty_1^{k_1} \quad | \quad \cdots \quad | \quad \mathsf{C}_m\ ty_m^1\ \ldots\ ty_m^{k_m}$$

Each type $(\alpha_1,\ldots,\alpha_n)rty$ specified by an equation of this form can be abstractly characterized by a single theorem of higher order logic. This theorem is the output of the HOL derived rule for defining types and has the following general form:

$$\begin{array}{l}\vdash \forall f_1\ \cdots\ f_m.\ \exists!fn{:}(\alpha_1,\ldots,\alpha_n)rty{\rightarrow}\beta.\\ \quad \forall x_1^1\cdots x_1^{k_1}.fn(\mathsf{C}_1\ x_1^1\ \ldots\ x_1^{k_1}) = f_1\ (fn\ x_1^1)\ \ldots\ (fn\ x_1^{k_1})\ x_1^1\ \ldots\ x_1^{k_1}\ \wedge\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\vdots\\ \quad \forall x_m^1\cdots x_m^{k_m}.fn(\mathsf{C}_m\ x_m^1\ \ldots\ x_m^{k_m}) = f_m\ (fn\ x_m^1)\ \ldots\ (fn\ x_m^{k_m})\ x_m^1\ \ldots\ x_m^{k_m}\end{array} \tag{19}$$

where the right hand sides of the equations include recursive applications $(fn\ x_i^j)$ of the function fn only for variables x_i^j of type $(\alpha_1,\ldots,\alpha_n)rty$.

Theorem (19) states that for any m functions f_1, ..., f_m there is a *unique* function fn which satisfies a 'primitive recursive' definition whose form is determined by the given functions f_1, ..., f_m. This is an abstract characterization of the type $(\alpha_1,\ldots,\alpha_n)rty$: it states the essential properties of the type, but does so without reference to the way it is represented. It follows from this theorem that every value of type $(\alpha_1,\ldots,\alpha_n)rty$ is constructed by one of the constructors C_1, ..., C_m, that each of these constructors is one-to-one, and that different constructors yield different values. The proof that theorem (19) implies these properties of $(\alpha_1,\ldots,\alpha_n)rty$ and the constructors C_1, ..., C_m can be outlined as follows.

The fact that every value of type $(\alpha_1,\ldots,\alpha_n)rty$ is constructed by one of the functions C_1, ..., C_m follows from the uniqueness part of theorem (19). Suppose there is some value, v say, such that $v \neq (\mathsf{C}_i\ x_i^1\ \ldots\ x_i^{k_i})$ for $1 \leq i \leq m$. I.e. v is not constructed by any C_i. One could then define two functions f and g of type $(\alpha_1,\ldots,\alpha_n)rty{\rightarrow}bool$ which yield the boolean T for all values constructed by any constructor C_i:

$$\forall x_i^1\cdots x_i^{k_i}.\ f(\mathsf{C}_i\ x_i^1\ \ldots\ x_i^{k_i}) = g(\mathsf{C}_i\ x_i^1\ \ldots\ x_i^{k_i}) = \mathsf{T} \qquad \text{for } 1 \leq i \leq m \tag{20}$$

and when applied to v yield different results: $f\ v = \mathsf{T}$ and $g\ v = \mathsf{F}$. If f and g are defined this way then $f \neq g$, since $f\ v \neq g\ v$. But from the uniqueness part of theorem (19) it follows that if f and g satisfy (20) then $f = g$. Therefore no such value v exists, and every value of type $(\alpha_1,\ldots,\alpha_n)rty$ is constructed by some C_i.

The fact that the constructors C_1, ..., C_m are one-to-one can be proved by using theorem (19) to define a 'destructor' function D_i for each C_i such that:

$$\vdash \mathsf{D}_i(\mathsf{C}_i\ x_i^1\ \ldots\ x_i^{k_i}) = (x_i^1,\ldots,x_i^{k_i})$$

For each constructor C_i, such a function can be defined by appropriately specializing the corresponding quantified variable f_i in theorem (19). From the property of the destructor D_i shown above, it is then easy to prove that:

$$\vdash (\mathsf{C}_i\ x_i^1\ \ldots\ x_i^{k_i}) = (\mathsf{C}_i\ y_i^1\ \ldots\ y_i^{k_i}) \supset (x_i^1 = y_i^1 \wedge \cdots \wedge x_i^{k_i} = y_i^{k_i})$$

which states that C_i is one-to-one, as desired.

Finally, the fact that different constructors yield different values can be proved by appropriately specializing the universally quantified functions $f_1, \ldots, f_m$ in theorem (19) to obtain a theorem asserting the proposition shown below:

$$\vdash \exists fn.\, \forall x_i^1\ \cdots\ x_i^{k_i}.\, fn(\mathsf{C}_i\ x_i^1\ \ldots\ x_i^{k_i}) = i \qquad \text{for } 1 \leq i \leq m$$

This states the existence of a function fn which yields the natural number i when applied to values constructed by the ith constructor. This means that any two different constructors C_i and C_j yield different values of type $(\alpha_1, \ldots, \alpha_n)rty$, since applying fn to these values gives different natural numbers.

Using theorems of the form illustrated by (19) to axiomatize recursive types is closely related to the initial algebra approach to the theory of abstract data types [2, 5]. This approach is very elegant from a theoretical point of view, but it is also of *practical* value in the HOL mechanization of recursive type definitions. Each recursive type is characterized by a single theorem, and all the theorems which characterize such types have the same general form. This uniform treatment of recursive types is the basis for the *efficient* automation of their construction in HOL. It allows the axiom for any recursive type to be quickly derived from a pre-proved theorem stating that axioms of this kind hold for *all* such types. Furthermore, it makes it possible to derive useful standard properties of recursive types (e.g. structural induction) in a uniform way, with relatively short formal proofs and therefore by efficient programmed inference rules.

7.3 Constructing Representations for Recursive Types

This section outlines a method by which a representation can be found for any type specified by an equation of the form described in Section 7.1. Each representation is an appropriately-defined subset of a type constructed using the type constant *one*, the type operators $\times$ and $+$, and the type $(\alpha)Tree$. A simple example is first given in Sections 7.3.1 and 7.3.2; the method for finding representations in general is then outlined in Section 7.3.3.

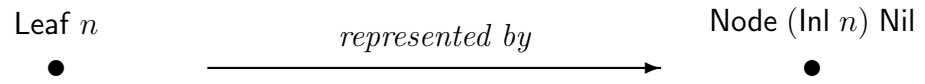
7.3.1 An Example: the Representation of Binary Trees

Consider the type *btree* described above in Section 7.1.1. This type was specified informally by:

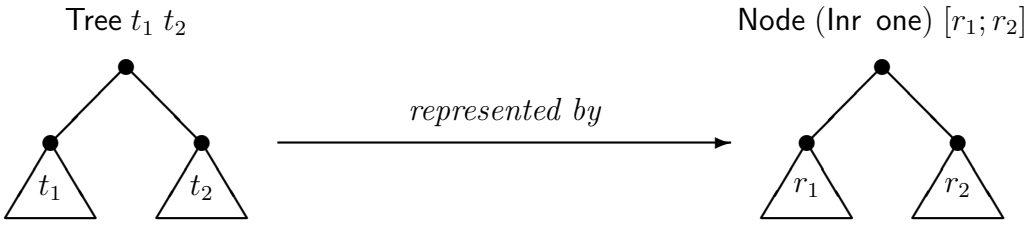
$$btree \quad ::= \quad \mathsf{Leaf}\ \ num \quad | \quad \mathsf{Tree}\ \ btree\ \ btree$$

The type *btree* specified by this equation can be represented in higher order logic by a subset of the set denoted by the compound type $(num + one)Tree$. This type denotes the set of all trees (of any shape) whose nodes are labelled either by a value of type *num* or by the single value one of type *one*. The idea of this representation is that each binary tree t of type *btree* is represented by a corresponding tree of type $(num + one)Tree$ which has both the same shape as t and the same labels on its nodes as t.

Consider, for example, the binary tree $(\mathsf{Leaf}\ n)$, consisting of a single leaf node labelled by the natural number n. This binary tree will be represented by a leaf node of type $(num + one)Tree$ labelled by the left injection $(\mathsf{Inl}\ n)$:



A binary tree (Tree t_1 t_2) which is not a leaf node, but has two subtrees t_1 and t_2, will be represented by a tree of type $(num + one)Tree$ which also has two subtrees and is labelled by the right injection (Inr one):



where r_1 and r_2 are the representations of the two binary trees t_1 and t_2 respectively. The 'dummy' value (Inr one) is used in this case to label the root node of the representation, since the corresponding binary tree being represented has no value associated with its root node.

7.3.2 Defining the Subset Predicate for *btree*

To introduce a type definition axiom for *btree*, a predicate Is_btree_REP must first be defined which is true of just those values of type $(num + one)Tree$ which represent binary trees using the scheme outlined above. This predicate is defined formally by building it up from two auxiliary predicates: Is_Leaf and Is_Tree. These two auxiliary predicates correspond to the two kinds of binary trees which will be represented, and each one states what the representation of the corresponding kind of binary tree looks like.

The predicates Is_Leaf and Is_Tree are defined as follows. Every value in the representation is a tree of the form (Node v tl), where v is a label of type $(num + one)$ and tl is a list of subtrees. If such a tree represents a leaf node (Leaf n), then the label v must be the value (Inl n) and the list tl must be empty. These conditions are expressed formally by the predicate Is_Leaf, defined as follows:

$$\vdash \mathsf{Is\_Leaf}\ v\ tl = (\exists n.\ v = \mathsf{Inl}\ n \wedge \mathsf{Length}\ tl = 0)$$

If (Node v tl) represents a binary tree (Tree t_1 t_2) with two subtrees, then the list of subtrees tl must have length two, and the label v must be the value (Inr one). The definition of Is_Tree is therefore:

$$\vdash \mathsf{Is\_Tree}\ v\ tl = (v = \mathsf{Inr\ one} \wedge \mathsf{Length}\ tl = 2)$$

The two predicates Is_Leaf and Is_Tree state what kind of values v and tl must be for the tree (Node v tl) to be the *root* node of legal binary-tree representation. But if a general tree of type $(num + one)Tree$ in fact represents a binary tree, then not only its root node but every node it contains (i.e. all its subtrees) must also satisfy either Is_Leaf or Is_Tree. This can be expressed formally in logic by first defining a higher order function All recursively on trees as follows:

$$\vdash \mathsf{All}\ P\ (\mathsf{Node}\ v\ tl) = P\ v\ tl \wedge \mathsf{Every}\ (\mathsf{All}\ P)\ tl$$

Using All, the predicate Is_btree_REP can then be defined such that it is true of a tree t exactly when the label and subtree list of every node in t satisfies either Is_Leaf or Is_Tree. The definition of Is_btree_REP is simply:

$$\vdash \mathsf{Is\_btree\_REP}\ t = \mathsf{All}\ \ (\lambda v.\, \lambda tl.\, \mathsf{Is\_Leaf}\ v\ tl\ \vee\ \mathsf{Is\_Tree}\ v\ tl)\ \ t$$

This predicate exactly specifies the subset of $(num + one)Tree$ whose values represent binary trees, and can therefore be used to introduce a type definition axiom for the new type *btree* in the usual way. All the predicates which specify representations of recursive type are defined using All in exactly the way shown above for Is_btree_REP.

7.3.3 Finding Representations in General

The representation of binary trees by a subset of $(num + one)Tree$ illustrates the general method for finding representations of any type specified by an equation of the form described in Section 7.1. In general, a recursive type specified by an equation of this kind denotes a set of labelled *trees* with a fixed number of different kinds of nodes. Any such type can therefore be represented by a subset of values denoted by some instance of the defined type $(\alpha)Tree$ of general trees.

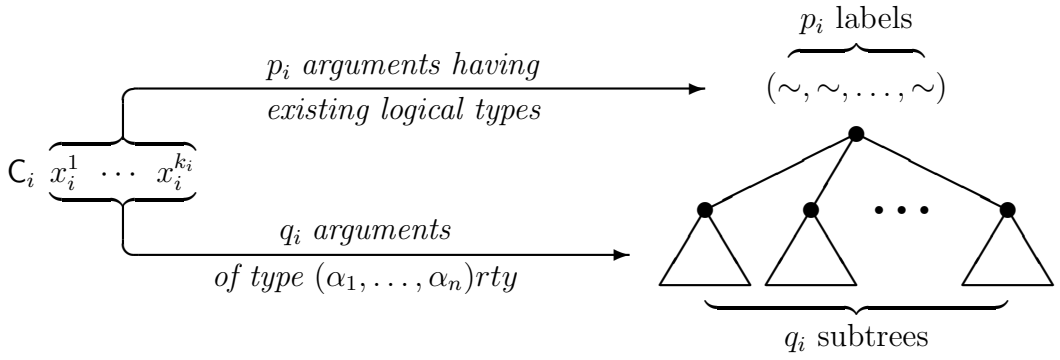
Suppose, for example, that $(\alpha_1, \ldots, \alpha_n)rty$ is specified by:

$$(\alpha_1, \ldots, \alpha_n)rty \quad ::= \quad \mathsf{C}_1\ ty_1^1\ \ldots\ ty_1^{k_1} \quad | \quad \cdots \quad | \quad \mathsf{C}_m\ ty_m^1\ \ldots\ ty_m^{k_m}$$

This equation specifies a type with m different kinds of values, corresponding to the m constructors C_1, ..., C_m. When this type is defined formally in higher order logic, each of its values will be denoted by some term of the form:

$$\mathsf{C}_i\ x_i^1\ \ldots\ x_i^{k_i}$$

where C_i is a constructor and each argument x_i^j is a value of type ty_i^j for $1 \leq j \leq k_i$. In the general case of a recursive type, some of the k_i arguments to C_i will have existing logical types and some will have the type $(\alpha_1, \ldots, \alpha_n)rty$ itself. Let p_i be the number of arguments which have existing logical types and let q_i be the number of arguments which have type $(\alpha_1, \ldots, \alpha_n)rty$, where $k_i = p_i + q_i$. The abstract value of type $(\alpha_1, \ldots, \alpha_n)rty$ denoted by $\mathsf{C}_i\ x_i^1\ \ldots\ x_i^{k_i}$ can be represented by a tree which has q_i subtrees and p_i values associated with its root node. This is illustrated by the diagram shown below:



In the general case illustrated by this diagram, the tree representing $\mathsf{C}_i\ x_i^1\ \ldots\ x_i^{k_i}$ is labelled by p_i-tuple of values. Each of these values is one of the p_i arguments to C_i which are not of type $(\alpha_1, \ldots, \alpha_n)rty$ but have types which already exist in the logic. When $p_i = 0$, the representing tree is labelled not by a tuple but by the constant one (as was done for the constructor Tree of $btree$). And when $p_i = 1$ the representing tree is labelled simply by a single value of the appropriate type (as was done for the constructor Leaf of $btree$). The q_i subtrees shown in the diagram are the representations of the arguments to C_i which have the type $(\alpha_1, \ldots, \alpha_n)rty$. If $q_i = 0$ then the representing tree has no subtrees.

Each of the m kinds of values constructed by C_1, ..., C_m can be represented by a tree using the scheme outlined above. In general, a value obtained using the ith constructor C_i will be represented by a tree labelled by a tuple of p_i values. The representing type for $(\alpha_1, \ldots, \alpha_n)rty$ will therefore be a type expression of the form:

$$(\overbrace{(\underbrace{ty\ \times\ \cdots\ \times\ ty}_{\text{product of } p_1 \text{ types}})\ +\ \cdots\ +\ (\underbrace{ty\ \times\ \cdots\ \times\ ty}_{\text{product of } p_m \text{ types}})}^{\text{sum of } m \text{ products}})Tree$$

where the ty's are the existing logical types occurring in the equation which specifies the new type $(\alpha_1, \ldots, \alpha_n)rty$ being defined.

Using this scheme, a predicate $\mathsf{Is\_rty\_REP}$ can be defined to specify a set of trees to represent $(\alpha_1, \ldots, \alpha_n)rty$ in exactly the same way as the predicate $\mathsf{Is\_btree\_REP}$ was defined for the representation of $btree$. The definition of $\mathsf{Is\_rty\_REP}$ will have the form:

$$\vdash \mathsf{Is\_rty\_REP}\ t = \mathsf{All}\ \ (\lambda v.\, \lambda tl.\, \mathsf{Is\_C}_1\ v\ tl\ \vee\ \cdots\ \vee\ \mathsf{Is\_C}_m\ v\ tl)\ \ t$$

where each $\mathsf{Is\_C}_i$ is an auxiliary predicate specifying which trees represent values constructed by the corresponding constructor C_i. The ith auxiliary predicate $\mathsf{Is\_C}_i$ is defined as follows. When $i \neq m$, the definition is:

$$\vdash \mathsf{Is\_C}_i\ v\ tl = \exists x_1 \ldots x_{p_i}.\, v = \mathsf{Inl}(\underbrace{\mathsf{Inr} \cdots (\mathsf{Inr}}_{i-1\ \mathsf{Inr's}}(x_1, \ldots, x_{p_i})) \cdots) \wedge \mathsf{Length}\ tl = q_i$$

where p_i is the number of arguments to C_i which have existing logical types, and q_i is the number of arguments of type $(\alpha_1, \ldots, \alpha_n)rty$. This definition states that if a tree $(\mathsf{Node}\ v\ tl)$ represents a value $\mathsf{C}_i\ x_i^1\ \ldots\ x_i^{k_i}$ then it must have the right number subtrees in tl and its label v must be an appropriate injection of some p_i-tuple (of the right logical type, of course). When $i = m$, the definition is similar:

$$\vdash \mathsf{Is\_C}_m\ v\ tl = \exists x_1 \ldots x_{p_m}.\, v = (\underbrace{\mathsf{Inr} \cdots (\mathsf{Inr}}_{m-1\ \mathsf{Inr's}}(x_1, \ldots, x_{p_m})) \cdots) \wedge \mathsf{Length}\ tl = q_m$$

The only difference is that the last injection applied is Inr, not Inl.

7.4 Deriving Abstract Axioms for Recursive Types

The *uniform* treatment of representations for recursive types makes it possible to write an efficient HOL derived inference rule which proves abstract axioms for them efficiently. Every representation is some subset '$\mathsf{All}\ P$' of an instance of $(\alpha)Tree$. A general theorem can therefore be formulated stating that an abstract axiom of the

required form holds for *any* recursive type represented this way. This theorem can then be simply 'instantiated' to obtain an abstract axiom for any particular recursive type.

The theorem stating that every recursive type satisfies an abstract axiom of the desired form is shown below:

$$\begin{array}{l}\vdash \forall P.\ \forall Abs{:}(\alpha)Tree{\to}\beta.\ \forall Rep{:}\beta{\to}(\alpha)Tree.\\ \quad (\forall a.\ Abs(Rep\ a){=}a \wedge \forall r.\ \mathsf{All}\ P\ r = (Rep(Abs\ r){=}r)) \supset\\ \quad \forall f.\ \exists!\ fn.\ \forall v\ tl.\ P\ v\ (\mathsf{Map}\ Rep\ tl) \supset\\ \qquad\qquad fn(Abs(\mathsf{Node}\ v\ (\mathsf{Map}\ Rep\ tl))) = f\ (\mathsf{Map}\ fn\ tl)\ v\ tl\end{array} \tag{21}$$

Informally, this theorem states that any type β which is represented by (i.e. is isomorphic to) a set '$\mathsf{All}\ P$' of trees satisfies an abstract axiom of the form described in Section 7.2. Theorem 21 makes this assertion in form of an implication:

$$\vdash \forall P. \cdots \langle \beta \text{ is isomorphic to '}\mathsf{All}\ P\text{'}\rangle \supset \langle \text{abstract axiom for } \beta\rangle$$

where the antecedent of this implication is written formally as follows:

$$\forall a.\ Abs(Rep\ a){=}a \wedge \forall r.\ \mathsf{All}\ P\ r = (Rep(Abs\ r){=}r)$$

This simply says that β is isomorphic to the set of trees of type $(\alpha)Tree$ which satisfy $\mathsf{All}\ P$. The type variable β stands for the new recursive type which is represented by $\mathsf{All}\ P$, and the variables *Abs* and *Rep* are the abstraction and representation functions for β.

The conclusion of theorem (21) states that functions can be uniquely defined by 'primitive recursion' on the structure which β inherits from $\mathsf{All}\ P$. That is, for any f, there is a unique function $fn{:}\beta{\to}\gamma$ which satisfies the recursive equation:

$$fn(Abs(\mathsf{Node}\ v\ (\mathsf{Map}\ Rep\ tl))) = f\ (\mathsf{Map}\ fn\ tl)\ v\ tl$$

whenever the condition $P\ v\ (\mathsf{Map}\ Rep\ tl)$ holds of v and tl. This condition on v and tl restricts the recursive equation shown above to apply only to 'well-constructed' values of type β. If $P\ v\ (\mathsf{Map}\ Rep\ tl)$ holds, then $\mathsf{All}\ P$ is true of the value $\mathsf{Node}\ v\ (\mathsf{Map}\ Rep\ tl)$ on the left hand side of the equation. The corresponding *abstract* value, denoted by:

$$Abs(\mathsf{Node}\ v\ (\mathsf{Map}\ Rep\ tl)),$$

will then be a correctly-represented value of type β. The example in Section 7.4.1 below shows how the form of the predicate P in the condition $P\ v\ (\mathsf{Map}\ Rep\ tl)$ determines the final 'shape' of the resulting axiom.

Theorem (21) illustrates the expressive power which higher-order variables and type polymorphism give to higher order logic. The variable P in this theorem ranges (essentially) over all predicates on $(\alpha)Tree$. And the two type variables α and β can be instantiated to any two logical types. Theorem (21) therefore asserts that an abstract axiom holds for *any* recursive type, since any such type is isomorphic to an appropriate subset $\mathsf{All}\ P$ of some instance of (α)Tree. Because general results like theorem (21) can be formulated as theorems in the logic, they can be used to make programmed inference rules in HOL efficient. Derived inference rules can use such pre-proved general theorems to avoid having to do costly 'run time' inference. Theorem (21) is used in this way by the derived rule which automates recursive type definitions.

The example given in the following section shows how this derived rule uses the general theorem (21) to prove the abstract axiom for a particular recursive type.

7.4.1 Example: Deriving the Axiom for *btree*

The example given in this section is the proof of the abstract axiom for *btree*, the type whose representation was described in Section 7.3.2. The following is the sequence of main steps which the HOL system carries out to define *btree* and derive an abstract axiom for it:

(1) Define the subset predicate Is_btree_REP, introduce a type definition axiom for *btree*, and define the associated abstraction and representation functions ABS:$(num+one)Tree \rightarrow btree$ and REP:$btree \rightarrow (num+one)Tree$.

This is done as outlined in Sections 7.3.2 and 3.3. The result of this step is the two theorems shown below:

$$\vdash \forall a.\, \mathsf{ABS}(\mathsf{REP}\ a) = a$$
$$\vdash \forall r.\, \mathsf{All\ Is\_btree\_REP}\ r = (\mathsf{REP}(\mathsf{ABS}\ r) = r))$$

These theorems simply state that the newly-introduced type constant *btree* denotes a set of values which is isomorphic to the subset of $(num+one)Tree$ defined by All Is_btree_REP.

(2) Use theorem (21) to obtain an (unsimplified) abstract axiom for *btree*.

If the type variables α and β in theorem (21) are instantiated to $(num + one)$ and *btree* respectively, then the universally quantified variables P, *Abs*, and *Rep* can be specialized to Is_btree_REP, ABS, and REP. The resulting instance of theorem (21) is an implication whose antecedent matches the two theorems about ABS and REP derived in the previous step. The theorem shown below therefore follows simply by modus ponens (and rewriting, with the definition of Is_btree_REP):

$$\vdash \forall f.\, \exists! fn.\, \forall v\ tl.\, (\mathsf{Is\_Leaf}\ v\ (\mathsf{Map\ REP}\ tl) \vee \mathsf{Is\_Tree}\ v\ (\mathsf{Map\ REP}\ tl)) \supset fn(\mathsf{ABS}(\mathsf{Node}\ v\ (\mathsf{Map\ REP}\ tl))) = f\ (\mathsf{Map}\ fn\ tl)\ v\ tl$$

This theorem expresses the essence of the desired abstract axiom for *btree*. The remaining steps carried out by the system are sequence of straightforward simplifications of this theorem which put it into the desired final form.

(3) Remove the disjunction: Is_Leaf v (Map REP tl) $\vee$ Is_Tree v (Map REP tl).

The theorem derived in the previous step contains a term which has the form $\forall v\, tl.\, (P \vee Q) \supset R$. By a simple proof in predicate calculus, this term is equivalent to the conjunction: $(\forall v\, tl.\, P \supset R) \wedge (\forall v\, tl.\, Q \supset R)$. The theorem derived in the previous step is therefore equivalent to:

$$\begin{aligned}\vdash \forall f.\, \exists! fn.\, \forall v\ tl.\, &\mathsf{Is\_Leaf}\ v\ (\mathsf{Map\ REP}\ tl) \supset \\ &fn(\mathsf{ABS}(\mathsf{Node}\ v\ (\mathsf{Map\ REP}\ tl))) = f\ (\mathsf{Map}\ fn\ tl)\ v\ tl\ \wedge \\ \forall v\ tl.\, &\mathsf{Is\_Tree}\ v\ (\mathsf{Map\ REP}\ tl) \supset \\ &fn(\mathsf{ABS}(\mathsf{Node}\ v\ (\mathsf{Map\ REP}\ tl))) = f\ (\mathsf{Map}\ fn\ tl)\ v\ tl\end{aligned}$$

In the general case of a type with m constructors, the subset predicate contains

a disjunction of the general form:

$$\mathsf{Is\_C}_1\ v\ (\mathsf{Map}\ Rep\ tl)\ \vee\ \cdots\ \vee\ \mathsf{Is\_C}_m\ v\ (\mathsf{Map}\ Rep\ tl)$$

When this step is done, it will introduce a conjunction of m implications in the body of the abstract axiom, each of which corresponds to one of the m constructors $\mathsf{C}_1, \ldots, \mathsf{C}_m$.

(4) Rewrite with the definitions of Is_Leaf and Is_Tree. This yields:

$$\begin{aligned}\vdash \forall f.\, \exists! fn.\, \forall v\ tl.\, &(\exists n.\, v = \mathsf{Inl}\ n \wedge \mathsf{Length}(\mathsf{Map\ REP}\ tl) = 0) \supset \\ &fn(\mathsf{ABS}(\mathsf{Node}\ v\ (\mathsf{Map\ REP}\ tl))) = f\ (\mathsf{Map}\ fn\ tl)\ v\ tl\ \wedge \\ \forall v\ tl.\, &(v = \mathsf{Inr\ one} \wedge \mathsf{Length}(\mathsf{Map\ REP}\ tl) = 2) \supset \\ &fn(\mathsf{ABS}(\mathsf{Node}\ v\ (\mathsf{Map\ REP}\ tl))) = f\ (\mathsf{Map}\ fn\ tl)\ v\ tl\end{aligned}$$

Note: In the HOL implementation, the predicates Is_Leaf and Is_Tree are not actually defined as new constants; they are instead written using λ-terms. This step therefore does not need to be done in the HOL implementation.

(5) Simplify terms of the form: $\mathsf{Length}(\mathsf{Map\ REP}\ tl) = m$.

A term of the form $\mathsf{Length}(\mathsf{Map\ REP}\ tl) = m$ is equivalent to a simplified term of the form $\mathsf{Length}\ tl = m$. This in turn is equivalent to saying that tl is equal to some list of m values: $\exists t_1 \ldots t_m.\, tl = [t_1; \ldots; t_m]$. The terms involving Length in the previous theorem can therefore be simplified, resulting in the following theorem:

$$\begin{aligned}\vdash \forall f.\, \exists! fn.\, \forall v\ tl.\, &(\exists n.\, v = \mathsf{Inl}\ n \wedge tl = \mathsf{Nil}) \supset \\ &fn(\mathsf{ABS}(\mathsf{Node}\ v\ (\mathsf{Map\ REP}\ tl))) = f\ (\mathsf{Map}\ fn\ tl)\ v\ tl\ \wedge \\ \forall v\ tl.\, &(v = \mathsf{Inr\ one} \wedge \exists t_1\ t_2.\, tl = [t_1; t_2]) \supset \\ &fn(\mathsf{ABS}(\mathsf{Node}\ v\ (\mathsf{Map\ REP}\ tl))) = f\ (\mathsf{Map}\ fn\ tl)\ v\ tl\end{aligned}$$

This step introduces the variables t_1 and t_2. They range over values of type *btree* and occur in the axiom for *btree* in its final form.

(6) Remove equations of the form: $v = \cdots$ and $tl = \cdots$.

The antecedents of the two logical implications in the previous theorem both contain equations giving values for v and tl. These can be removed by using (a generalization of) the fact that in predicate calculus a term of the form $\forall y.\, (\exists x.\, y = tm_1[x]) \supset tm_2[y]$ is equivalent to $\forall x.\, tm_2[tm_1[x]]$. The result of removing the equations for v and tl is:

$$\begin{aligned}\vdash \forall f.\, \exists! fn.\, \forall n.\, &fn(\mathsf{ABS}(\mathsf{Node}\ (\mathsf{Inl}\ n)(\mathsf{Map\ REP\ Nil}))) \\ &\quad = f\ (\mathsf{Map}\ fn\ \mathsf{Nil})\ (\mathsf{Inl}\ n)\ \mathsf{Nil}\quad \wedge \\ \forall t_1\ t_2.\, &fn(\mathsf{ABS}(\mathsf{Node}\ (\mathsf{Inr\ one})\ (\mathsf{Map\ REP}\ [t_1; t_2]))) \\ &\quad = f\ (\mathsf{Map}\ fn\ [t_1; t_2])\ (\mathsf{Inr\ one})\ [t_1; t_2]\end{aligned}$$

The body of the theorem now consists of two equations. These define the value of fn for the two different kinds of binary trees.

(7) Rewrite with the definition of Map. This yields:

$$\begin{aligned}\vdash \forall f.\, \exists! fn.\, \forall n.\, fn(\mathsf{ABS}(\mathsf{Node}\ (\mathsf{Inl}\ n)\ \mathsf{Nil})) \\ = f\ \mathsf{Nil}\ (\mathsf{Inl}\ n)\ \mathsf{Nil} \quad \wedge \\ \forall t_1\, t_2.\, fn(\mathsf{ABS}(\mathsf{Node}\ (\mathsf{Inr\ one})\ [\mathsf{REP}\ t_1; \mathsf{REP}\ t_2])) \\ = f\ [fn\ t_1; fn\ t_2]\ (\mathsf{Inr\ one})\ [t_1; t_2]\end{aligned}$$

(8) Define the abstract constructors Leaf and Tree as follows:

$$\begin{aligned}\vdash \mathsf{Leaf}\ n &= \mathsf{ABS}(\mathsf{Node}\ (\mathsf{Inl}\ n)\ \mathsf{Nil}) \\ \vdash \mathsf{Tree}\ t_1\ t_2 &= \mathsf{ABS}(\mathsf{Node}\ (\mathsf{Inr\ one})\ [\mathsf{REP}\ t_1; \mathsf{REP}\ t_2])\end{aligned}$$

The constructors Leaf and Tree defined by these equations first use Node to construct the representations of the required values and then use ABS to obtain the corresponding values of type *btree*. Rewriting the theorem derived in the previous step with these definitions yields:

$$\begin{aligned}\vdash \forall f.\, \exists! fn.\, \forall n.\, fn(\mathsf{Leaf}\ n) = f\ \mathsf{Nil}\ (\mathsf{Inl}\ n)\ \mathsf{Nil}\ \wedge \\ \forall t_1\, t_2.\, fn(\mathsf{Tree}\ t_1\ t_2) = f\ [fn\ t_1; fn\ t_2]\ (\mathsf{Inr\ one})\ [t_1; t_2]\end{aligned}$$

(9) Introduce two functions f_1 and f_2 in place of f.

With an appropriate choice of value for the universally quantified variable f, two functions f_1 and f_2 can be introduced for the right hand sides of the two equations. These define the value of fn separately for the two constructors Leaf and Tree. Specializing f to the appropriate function, and simplifying, gives:

$$\begin{aligned}\vdash \forall f_1\, f_2.\, \exists! fn.\, \forall n.\, fn(\mathsf{Leaf}\ n) = f_1\ n \wedge \\ \forall t_1\, t_2.\, fn(\mathsf{Tree}\ t_1\ t_2) = f_2\ (fn\ t_1)\ (fn\ t_2)\ t_1\ t_2\end{aligned}$$

This theorem is the abstract axiom for *btree*—in its final form.

The HOL derived rule which automates recursive type definitions carries out the sequence of steps shown above for each informal type specification entered by the user. An appropriate instance of theorem (21) yields an 'unsimplified' abstract axiom for the type being defined. This axiom is then systematically transformed into the form described in Section 7.2 by the sequence of simple equivalence-preserving steps shown above. The amount of actual logical inference that must be carried out is relatively small, and each step is a straightforward transformation of the theorem derived in the previous step. The HOL implementation of this procedure is therefore both efficient and robust.

8 Concluding Remarks

The method for defining recursive types described in Section 7 is the logical basis for a set of efficient theorem-proving tools the HOL system. In addition to the derived inference rule which automates recursive type definitions, a number of related tools have been implemented in HOL for generating proofs involving recursive types. These include:

- an inference rule which derives structural induction for recursive types, and related tools for interactively generating proofs by structural induction (e.g. a general structural induction *tactic*),
- a set of rules which automate the inference necessary to define functions by 'primitive recursion' on recursive types,
- derived rules which prove that the constructors of recursive types are one-to-one and yield distinct values, and
- tools for generating interactive proofs by case analysis on the constructors of recursive types.

Preliminary work is underway to extend these tools to deal with mutually recursive types, and types with equational constraints.

Defining a logical type in HOL is rarely the primary goal of the user of the system, but often a necessary part of some more interesting proof. The efficient automation of type definitions in HOL is therefore of significant practical value, since defining types 'by hand' in the system is tedious and tricky. The mechanization of type definitions described in this paper allows new recursive types to be introduced by the HOL user quickly and easily. This is made possible by the systematic construction of representations for these types, the uniform treatment of abstract axioms for them (using essentially the initial algebra approach to type specifications), and the expressive power of higher order logic itself.

Acknowledgements

Thanks are due to Albert Camilleri, Inder Dhingra and Mike Gordon for helpful comments on drafts of this paper, and to Thomas Forster for useful discussions about the construction of trees. I am grateful to Gonville and Caius College Cambridge for support in the form of an unofficial fellowship, during which the work described in this paper was done.

References

[1] Bird, R., and Wadler, P, *Introduction to Functional Programming*, Prentice Hall International Series in Computer Science (Prentice Hall, 1988).

[2] Burstall, R., and Goguen, J., 'Algebras, theories and freeness: an introduction for computer scientists', in: *Theoretical Foundations of Programming Methodology*, edited by M. Wirsing and G. Schmidt, Proceedings of the 1981 Marktoberdorf NATO Summer School, NATO ASI Series, Vol. C91 (Reidel, 1982), pp. 329–350.

[3] Church, A., 'A Formulation of the Simple Theory of Types', *Journal of Symbolic Logic*, Vol. 5 (1940), pp. 56–68.

[4] Cousineau, G., G. Huet, and L. Paulson, *The ML Handbook*, INRIA, (1986).

[5] Goguen, J.A., J.W. Thatcher, and E.G. Wagner, 'An initial algebra approach to the specification, correctness, and implementation of abstract data types', in: *Current Trends in Programming Methodology*, edited by R.T. Yeh (Prentice-Hall, New Jersey, 1978), IV, pp. 80–149.

[6] Gordon, M., 'HOL: A Machine Oriented Formulation of Higher Order Logic', Technical Report No. 68, Computer Laboratory, The University of Cambridge, Revised version (July 1985).

[7] Gordon, M.J.C., 'HOL: A Proof Generating System for Higher-Order Logic', in: *VLSI Specification, Verification and Synthesis*, edited by G. Birtwistle and P.A. Subrahmanyam, Kluwer International Series in Engineering and Computer Science, SECS35 (Kluwer Academic Publishers, Boston, 1988), pp. 73–128.

[8] Gordon, M.J., R. Milner, and C.P. Wadsworth, 'Edinburgh LCF: A Mechanised Logic of Computation', Lecture Notes in Computer Science, Vol. 78 (Springer-Verlag, Berlin, 1979).

[9] Harper, R., D. MacQueen, and R. Milner, Standard ML, Report No. ECS-LFCS-86-2, Laboratory for Foundations of Computer Science, Department of Computer Science, The University of Edinburgh (March 1986).

[10] Leisenring, A.C., *Mathematical Logic and Hilbert's ε-Symbol*, University Mathematical Series (Macdonald & Co., London, 1969).

[11] Milner, R., 'A Theory of Type Polymorphism in Programming', *Journal of Computer and System Sciences*, No. 17 (1978).

[12] Paulson, L.C., *Logic and Computation: Interactive Proof with Cambridge LCF*, Cambridge Tracts in Theoretical Computer Science 2 (Cambridge University Press, Cambridge, 1987).

Index

β-reduction, 3

Camilleri, A.J., 45
cartesian product type, 12–15, 35

Dhingra, I.S., 45
disjoint sum type, 15–18, 35

Forster, T., 45
Fourman, M., 7

Gordon, M., 2, 5, 7, 45

higher order logic, 2, 3
    types in, 1, 3, 4
Hilbert's ε-operator, 4, 5
HOL, 1–3, 5, 6, 8, 32, 33, 35, 36, 40–42, 44, 45
    derived inference rules in, 6, 41

initial algebra, 34, 36, 45

λ-calculus, 3
LCF, 2, 5
lists, 22–24, 34

mathematical induction, 21
Milner, R., 4
ML, 5, 6, 33

natural numbers, 19–22, 34

Peano's postulates, 20, 21, 24, 27
predicate calculus notation, 2
primitive recursion, 3, 21, 22, 35, 41, 44
    on labelled trees, 31
    on lists, 23, 25
    on trees, 27

Russell's paradox, 3

Standard ML, 32
structural induction, 36, 44
    on lists, 24
    on trees, 27

tactic, 44
trees, 24–29
    binary, 34, 37, 38, 41–44
    labelled, 29–32, 38
type constants, 4, 7, 34
type definition axioms, 7–9, 33
type definitions, 6–9
    automating, 1, 2, 7, 32, 33, 44, 45
type inference, 4
type operators, 4, 7–9, 34
type variables, 3, 4, 6, 8, 12, 14, 41
types
    abstract axioms for, 6, 7, 35, 36
    enumerated, 35
    record, 35
    recursive, 7, 24, 32, 34
    representation of, 7, 38–40