University of Edinburgh

Department of Computer Science

Automating Mutually Recursive Type Definitions In HOL

4th Year Project Report

Department of Artificial Intelligence

Claudio V. Russo

June 3, 1992

Abstract

This report identifies a general method for defining mutually-recursive concrete types and deriving their abstract axiomatisations in higher-order-logic. The method is designed to preserve consistency and is tailored to the implementation of an efficient, automated type definition facility for the interactive theorem prover HOL90.

Contents

Chapter 1

Higher Order Logic and its Implementation in the HOL System

The HOL system is an interactive theorem prover for Higher Order Logic. The HOL logic may be viewed as an extension of the predicate calculus with the following features:

- Variables can range over functions and predicates.
- Functions can take functions as arguments and return functions as results.
- Terms which denote functions are created using the notation of the *lambda calculus*.

Terms of the logic are *typed*. Types can be thought of as denoting sets of values and terms as denoting elements of these sets. In particular, logical formulae have type *bool*. Types come in three forms: type constants, type variables, and compound types which are constructed recursively from simpler type expressions.

The HOL system used for this project is implemented in Standard ML. Theorems of the logic are represented by the ML abstract data type `thm`. The only means of constructing theorems is by the application of inference rules to existing theorems, of which a small, consistent set are supplied as axioms. These inference rules form the interface to the `thm` type. The meta-language used in the HOL system is Standard ML itself, allowing fully general proof mechanization.

The aim of the project is to provide the first of a set of tools for reasoning about structured operational semantics. This application domain, like many others, requires a facility for formally defining the abstract syntax of a language from a given, informal description. For instance, a typical language, supporting both boolean and integer expressions, will allow boolean expressions to appear within an integer expression and vice versa, as in the following BNF fragment:

$$\begin{array}{lcl} Bexp & ::= & ... \mid Iexp \ = \ Iexp \mid ... \\ Iexp & ::= & ... \mid if \ \ Bexp \ \ then \ \ Iexp \ \ else \ \ Iexp \mid ... \end{array}$$

If the expressions of the language are to be regarded as terms of types *Bexp* and *Iexp* then the types must be defined by mutual recursion. The HOL system is supplied with a package for mechanically defining a restricted class of recursive types, but it does not deal with mutual recursion. Once a type has been created, a number of standard theorems for reasoning about this type can be derived automatically, such as a theorem stating the validity of structural induction. The project's aim is to extend this package to provide an efficient, effective and *consistency preserving* tool for extending HOL theories to encompass mutually recursive types.

1.1 The Syntax of HOL Terms

Blurring the distinction between Higher Order Logic and its implementation in the HOL system, we adopt the syntax of the latter for the remainder of this report. HOL incorporates the conventional logical operators of negation, conjunction, disjunction, implication and equivalence represented by `~`, `/\`, `\/`, `==>`, and `=` respectively. If `P` is a predicate taking an argument `x` then the universal and existential quantifiers `!`, `?` may be used to express: `!x. P x`, for every value of variable `x`, `P x` holds; `?x. P x`, for some value of variable `x`, `P x` holds. There is an additional quantifier `?!` such that `?!x. P x` has the meaning: there is a unique value for variable `x` such that `P x` holds. Nested quantification such as `!x1. !x2. .. !xn.tm` is abbreviated to `!x1 x2 .. xn.tm`.

To aid in the creation of functions, there is a conditional operator `(C => t1 | t2)` to express 'if C then t1 else t2'. The operator `o` denotes function composition, so that `(f o g) x = f(g(x))`. Functions can also be written using the notation of the λ-calculus. For example, the abstraction `\x. tm[x]` is a function of one argument. Simplifying the application `(\x. tm[x]) y` by β-reduction results in the term `tm[y/x]`, i. e. the body of the abstraction with all free occurrences of the variable `x` replaced by the term `y`.

Theorems in HOL are written as sequents. A sequent consists of a generally empty set of assumption formulae { `A1`, ... , `An` } separated by a turnstile `|-` from a conclusion formula `C`, and is often named for reference, e. g.:

```
TRIV [x=1, ... ,y=2] |- x+y = 3
```

1.2 Types and Their Syntax

Higher Order Logic is *typed*: every term of the logic must have a type that is consistent with that of its subterms. Ignoring polymorphism, a type may be thought of as denoting a set, and terms of the type as elements of the set. We write `tm:ty` to state that term `tm` has type `:ty`. The HOL system infers the most general type of a term from its context, and we will seldomly make this information explicit. Type expressions are built up from three syntactic classes: type constants, type variables and compound types, created by applying set building operations, or *type operators*, to argument types.

Type constants are identifiers which name fixed sets of values. The type `:one` is a singleton set, whose sole element is the constant (term) `one`. The two element set of truth values is `:bool` with constants `T` for *true* and `F` for *false*. Similarly, `:num` is the natural numbers with the infinite set of constants { 0,1,2, ... }.

Type variables, indicated by identifiers prefixed by a prime, e. g. `:'a`, stand for arbitrary type expressions. A type is polymorphic if it contains one or more type variables. A *substitution instance* of a polymorphic type is obtained by substituting types for one or more of its type variables. Note that a theorem containing type variables holds for its all substitution instances.

Compound types are built recursively form other type expressions by the application of type operators of the form `:(ty1, ... ,tyn)op`, where `:ty1` through `:tyn` are types and `op` is the n-ary type operator. In the expression `dom -> rng`, the infix operator `->` creates the space of all total functions from domain type `:rng` to range type `:rng`.

1.2.1 The Product of Two Types

The expression `:ty1 # ty2` denotes the cartesian product of types `:ty1` and `:ty2`. It is the set of ordered pairs whose first components are drawn from `:ty1` and whose second components are drawn from `:ty2`. Associated with the infix operator `#` are polymorphic functions for creating and dissecting values of the type. The (infix) injection function, or constructor `,` `:'a -> ('b -> ('a # 'b))` takes two arguments and produces their pair. The destructors `FST: ('a # 'b) -> 'a` and `SND: ('a # 'b) -> 'b` respectively project the first and second components out of a pair. The constants are related by the theorems:

```
FST |- !x y. FST (x,y) = x
SND |- !x y. SND (x,y) = y
PAIR_EQ  |- !x y a b. ((x,y) = (a,b)) = (x = a) /\ (y = b)
FORALL_PAIR_EQ |- !P:('a # 'b) -> bool. (!p. P p = !x y. P (x,y))
```

The theorem `PAIR_EQ` is useful for deriving component-wise equality from the equality of two pairs. `FORALL_PAIR_EQ` states that a predicate `P` holds for all pairs `p`, if and only if it holds for all pairs constructed by the `,` constant.

1.2.2 The Sum of Two Types

The expression `:ty1 + ty2` denotes the sum operation on types `:ty1` and `:ty2`. It can be viewed as the union of two disjoint sets: one obtained from `:ty1` by labeling each element as originating from the left summand; the other by labeling elements of `:ty2` as originating from the right. Values of type `:'a + 'b` are created by the left and right injection functions `INL:'a -> ('a + 'b)` and `INR:'b -> ('a + 'b)`. The corresponding projection functions are `OUTL:('a + 'b) -> 'a` and `OUTR:('a + 'b) -> 'b` related by:

```
OUTL |- !x. OUTL (INL x) = x)
OUTR |- !y. OUTR (INR y) = y)
```

Further theorems state that the constructors are one-one and that predicates over sums hold if and only if they hold over all injections:

```
INL_11 |- !x x'. (INL x = INL x') = (x = x')
INR_11 |- !y y'. (INR y = INR y') = (y = y')


FORALL_SUM_EQ |- !P:('a + 'b) -> bool.
                    (!s. P s) = (!x. P (INL x) /\ !y. P (INR y))
```

We also assume the existence of two infix functions `++` and `||`, called plus and sigma, respectively with types:

```
 ++ :('a -> 'c) -> ('b -> 'd) -> (('a + 'b) -> ('c + 'd))
 || :('a -> 'c) -> ('b -> 'c) -> (('a + 'b) -> c)
```

satisfying:

```
plus_DEF  |- !f g x. f ++ g (INL x) = INL (f x) /\
             !f g y. f ++ g (INR y) = INR (g y)


sigma_DEF |- !f g x. f || g (INL x) = f x /\
             !f g y. f || g (INR y) = g y
```

The function plus, `++`, takes three arguments: a function `f:'a -> 'c`, a function `g:'b -> 'd` and a sum `s:'a + 'b`. It examines the tag of `s` and applies `f` if it corresponds to a value of type `:'a` and `g` if it corresponds to a value of `:'b`. It then injects the result into the sum type `:'c + 'd` accordingly, mirroring the structure of the domain in the range.

The function sigma, `||`, behaves similarly yet requires `f` and `g` to share the same range type `:'c`. It maps directly from `:'a + 'b` to `:'c`. It follows directly from the definition of function composition, `o`, and the extensionality of functions that:

```
plus_o_THM |- !f g x. (f || g) o INL = f /\
              !f g y. (f || g) o INR = g
```

1.2.3 The Type Of Finite Length Lists

The set of finite length list of elements of type `:ty` is denoted by `:ty list`. Lists are manipulated by the familiar constructors and destructors:

```
NIL:'a list             CONS:'a -> 'a list -> 'a list
HD:'a list -> 'a        TL:'a list -> 'a list
```

As convenient syntactic sugar, lists are represented by semi-colon separated sequences of values enclosed by brackets, e. g. `NIL = []` and `CONS 1 (CONS 2 NIL) = [1;2]`. We will make use of the function `MAP:('a -> 'b) -> ('a list) -> ('b list)` defined by:

```
MAP |- !f. MAP f [] = [] /\
       !f h t. MAP f (CONS h t) = CONS (f h) (MAP f t)
```

We shall also refer to the higher-order predicate `EVERY` which takes a predicate `P` and and a list `l` and holds if and only if each element of the list satisfies `P`:

```
EVERY_DEF |-  !P. EVERY P [] = T  /\
              !P h t. EVERY P (CONS h t) = P h /\ EVERY P t
```

1.3 HOL Theories and Conservative Extension

A *theory* in the HOL logic is a collection of term constants, type constants and operators, axioms and theorems derived from these axioms. Theories can be extended by *defining* new constants and types in terms of existing ones and proving theorems about them in terms of their definitions, or by adding these theorems directly as axioms. The former method is tedious but ensures consistency, whilst the later is straightforward but provides no such guarantee.

1.3.1 Hilbert's Selection Operator

HOL incorporates Hilbert's ϵ-symbol, or *selection* operator, represented by `@` in the system. The operator binds a variable to a formula. The expression `@x:ty.P x` denotes an arbitrary but fixed value of type `:ty`. In particular, if `|- ?x. P x` is a theorem of the theory, then the denoted element satisfies the predicate `P:ty -> bool`, otherwise it simply denotes some value of type `:ty`. This is stated in HOL by the theorem:

```
SELECT_THM |-!P. (?x. P x) = P (@x.P x)
```

To avoid the possibility of creating expression which denote terms that do not exist, HOL requires that each type be inhabited by at least one element. This condition must in turn be satisfied by any new types added to the logic.

1.3.2 Theory Extension by Constant Definition

If `c` is an undeclared constant identifier and `tm` a term of the current theory, T, i. e. `tm` involves only constants and types of T, then a larger theory T' can be obtained by declaring c to be a new constant satisfying the definitional axiom `|- c = tm`. By the monotonicity of higher order logic, T' is a consistent theory if and only if T is. Thus definitional extension preserves consistency and is conservative.

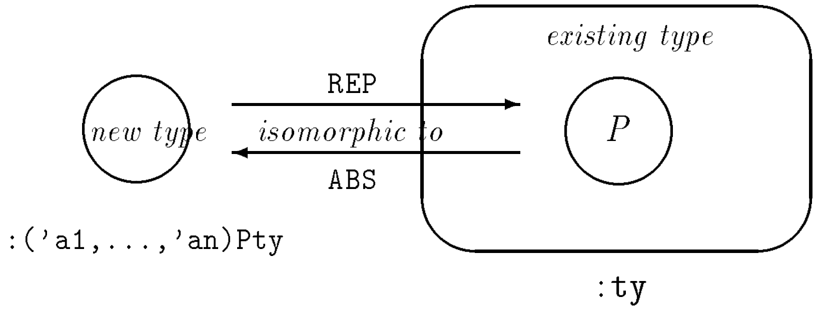

Figure 1.1: Adding a new type to HOL by conservative extension

1.3.3 Theory Extension by Constant Specification

Using the select operator and definitional extension, one can, given the knowledge that at least one value satisfying a predicate `P` exists, introduce a new constant to denote a fixed but arbitrary such value. Suppose `|- ?x. P x` is a theorem in the current theory. Then by a specialisation of `SELECT_THM` and Modus Ponens, so is `|- P (@x.P a)`. If one now defines a new constant `c` such that `|- c = @x.P x`, then substituting equals for equals in the previous theorem yields `|- P c`. The technique generalises to theorems with nested existential quantification: from the theorem `|- ?x y x. Q x y x` we can define constants `a`, `b` and `c` such that `|- Q a b c` holds in the extended theory. Whereas definitional extension fully specifies the constant in question, the method of constant specification only constrains the constant to denote any one of a number values satisfying the given predicate. Since it is a special form of definitional extension involving Hilbert's operator, it too preserves consistency.

1.3.4 Theory Extension by Type Definition

Figure 1.1 summarises the mechanism for adding new types to HOL. To define a new type operator `Pty` in HOL one must:

1. Identify an existing type `:ty` capable of representing elements of `:Pty`. The type `:Pty` is intended to be isomorphic to a subset P of `:ty`.

2. Describe the subset P by providing a *characteristic predicate* `P:ty -> bool`, mapping every ty value in P to `T` and and all others to `F`.

3. Prove that the set P is non-empty to ensure that `P` will describe a legal type (this proof obligation is due to the use of Hilbert's selection operator). The proof

amounts to showing `|- ?x:ty. P x`.

4. Asserting a *type definition axiom* stating that `:Pty` is isomorphic to the subset P of `:ty`. Note that the addition of this axiom preserves consistency.

The result of this last step is the addition of the type operator `('a1, ... ,'an)Pty` to the theory's syntax of types, where `'a1` through `'an` are the type variables appearing in the representing type `:ty`. The added axiom has the form:

```
TY_DEF |- ?rep:('a1, ... ,'an)Pty -> ty. TYPE_DEFINITION P rep
```

It states the existence of a representation function mapping from values of the new type to the representing type, satisfying the higher-order predicate `TYPE_DEFINITION P rep`, where:

```
TYPE_DEFINITION |- !P rep.
                          TYPE_DEFINITION P rep =
                            !x' x''. (rep x' = rep x'') ==> (x' = x'') /\
                            !x. P x = (?x'. x = rep x')
```

Hence `TY_DEF` states the existence of a representation function which is both one-to-one and onto the representing subset carved out of `:ty` by the characteristic predicate `P`. By the method of constant specification, one can now define a constant `REP:('a1, ... ,'an)Pty -> ty` to denote such a representation function and obtain:

```
REP_BIJECTIVE |- TYPE_DEFINITION P REP
```

Since any bijective function has a corresponding inverse, there should be an abstraction function of type `:ty -> ('a1, ... ,'an)Pty` mapping values from the representing subset back to their correct abstract values, as expressed by the theorem:

```
ABS_EXISTS_THM  |- !P rep.
                         TYPE_DEFINITION P rep ==>
                            ?abs.( !a. abs (rep a) = a     /\
                                   !r. P r = (rep (abs r) = r) )
```

Specialising `ABS_EXISTS_THM` to our `P` and `REP` and by Modus Ponens on `REP_BIJECTIVE` yields:

```
|- ?abs.  (!a. abs (REP a) = a) /\  (!r. P r = (REP (abs r) = r))
```

allowing the introduction of the new constant `ABS:ty -> ('a1, ... ,'an)Pty`. satisfying:

```
|- !a. ABS (REP a) = a
|- !r. P r = (REP (ABS r) = r)
```

The *constructors* of the `('a1, ... ,'an)Pty` can now be defined in terms of the abstraction and representation functions. Properties that hold of the representing subset P will also hold of the abstract type by virtue of the isomorphism between them. An abstract axiomatisation (i. e. a collection of theorems stating properties of the abstract type in terms of its constructors and destructors, without reference to its representation) can be derived by recourse to the properties of the representing type and the isomorphism.

1.4 Defining Mutually Recursive Types in HOL

The input to the type definition package should be an informal *type specification* of the following form:

$$\begin{array}{rcl} Aty & ::= & C \;\; ty_C^1 \ldots ty_C^c \mid \cdots \mid D \;\; ty_D^1 \ldots ty_D^d \\ & \vdots & \\ Bty & ::= & E \;\; ty_E^1 \ldots ty_E^e \mid \cdots \mid F \;\; ty_F^1 \ldots ty_F^f \end{array}$$

where each ty_X^j is either an existing logical type or is one of the type abbreviations Aty through Bty. This set of equations specifies the mutually recursive types $('a_1, \cdots, 'a_n)Aty, \ldots, ('a_1, \cdots, 'a_n)Bty$ where $\{'a_1, \ldots, 'a_n\}$ is the set of type variables appearing collectively in the equations. The function C denotes a constructor for type Aty with functionality : $ty_C^1 \rightarrow \ldots \rightarrow ty_C^c \rightarrow ('a_1, \ldots, 'a_n)Aty$. The constructors D,E and F are similarly described.

The logical types specified by the set of equations denote the sets of all values which can be finitely generated using the constructors C,D,E and F, where each constructor is one-to-one and any two different constructors yield different values. We adopt the convention that, in the application of a constructor to some argument values, those arguments with types in $('a_1, \cdots, 'a_n)Aty, \ldots, ('a_1, \cdots, 'a_n)Bty$ are termed as possessing a *recursive type*, otherwise they are said to be of an *existing type*.

Any set of types specified in this way can be characterised by a single theorem of the form:

$$
\begin{array}{ll}
\vdash & !\ f_C \cdots f_D \cdots f_E \cdots f_F. \\
& ?!\ FN : (('a_1, \ldots, 'a_n)Aty + \cdots + ('a_1, \ldots, 'a_n)Bty) \rightarrow ('A + \cdots + 'B). \\
& !\ x_1\ \cdots\ x_c\ .\ (OUTA \circ FN \circ INA)\ (C\ x_1 \ldots x_c) = \\
& \qquad f_C\ (OUT \circ FN \circ IN\ x_1) \ldots (OUT \circ FN \circ IN\ x_c)\ x_1\ \ldots\ x_c \qquad \bigwedge \\
& \vdots \\
& !\ x_1\ \cdots\ x_d\ .\ (OUTA \circ FN \circ INA)\ (D\ x_1 \ldots x_d) = \\
& \qquad f_D\ (OUT \circ FN \circ IN\ x_1) \ldots (OUT \circ FN \circ IN\ x_d)\ x_1\ \ldots\ x_d \qquad \bigwedge \\
& \vdots \\
& !\ x_1\ \cdots\ x_e\ .\ (OUTB \circ FN \circ INB)\ (E\ x_1 \ldots x_e) = \\
& \qquad f_E\ (OUT \circ FN \circ IN\ x_1) \ldots (OUT \circ FN \circ IN\ x_e)\ x_1\ \ldots\ x_e \qquad \bigwedge \\
& \vdots \\
& !\ x_1\ \cdots\ x_f\ .\ (OUTB \circ FN \circ INB)\ (F\ x_1 \ldots x_f) = \\
& \qquad f_F\ (OUT \circ FN \circ IN\ x_1) \ldots (OUT \circ FN \circ IN\ x_f)\ x_1\ \ldots\ x_f
\end{array}
$$

where the applications of $(OUT \circ FN \circ IN\ x_i)$ appear only for those constructor arguments x_i with recursive type. The generic IN and OUT are abbreviations for the appropriate *injection* of x_i into the sum domain of FN, and *projection* of the result of applying FN onto the appropriate component type of the range $:' A + \cdots +' B$ of FN. The theorem constrains all values of $: ('a_1, \ldots, 'a_n)Aty$ to be mapped to $:' A$, $\ldots$, and all values of $: ('a_1, \ldots, 'a_n)Bty$ to $:' B$. It states that a function FN on the sum of the recursive types is uniquely specified by providing *case functions* f_C, $\ldots$, f_D, $\ldots$,f_E,...and f_F to deal with values contructed by C, $\ldots$, D, $\ldots$, E, ...and F respectively. Note that each case function expects not only the values of the respective constructor's arguments, but also the result of the recursive application of FN to each argument with a recursive type. The theorem thus expresses the validity of obtaining functions by the method of *primitive recursive definition.*

Such a theorem is sufficient for the derivation of further theorems showing that:

- different constructors yield different values;

- the constructors are one-to-one;
- properties of the types can be proved by structural induction;
- any value of a particular recursive type must have been created by an application of one of its associated constructors;

and serves as a concise *abstract axiomatisation* of the types. It is *abstract* because it refrains from referring to the underlying representation of the types; and an *axiomatisation*, because it neatly encapsulates all their expected properties.

1.5 The Type of Labelled, Finite-Branching Trees

A set of mutually-recursive, concrete types specified in the syntax of Section 1.4 can be represented by an instance of the existing type `:'a ltree`, with the sole constructor:

```
Node:'a -> (('a ltree) list) -> ('a ltree)
```

Each node in an `:'a ltree` is labelled by some value of type `:'a` and can have an arbitrary but finite number of ordered subtrees of the same type `'a ltree`. The idea is that the labels on the nodes are used to:

- distinguish which of the mutually recursive types the tree represents a value of;
- identify which constructor was used to produce the abstract value with this representation;
- store the values of any constructor arguments with non-recursive type, i.e. those arguments with types other than those being defined.

Furthermore, for a node to actually represent some constructed value, its subtrees are constrained to correspond in number, order and structure to representations of the recursively typed arguments to the constructor indicated by the label.

1.6 From Labelled Trees to Recursive Types

The type `:'a ltree` satisfies the primitive recursion theorem:

```
ltree_Axiom |- !f. ?!fn. !v tl. fn (Node v tl) = f (MAP fn tl) v tl
```

As a consequence, certain inductively defined subsets of `:'a ltree` satisfy their own primitive recursion theorems. Any type defined to be isomorphic to such a subset, inherits the property of possessing a primitive recursion theorem.

Consider the higher-order predicate `TRP` defined by:

```
TRP : ('a -> ('a ltree list) -> bool) -> ('a ltree) -> bool
TRP |- !P v tl. TRP P (Node v tl) = P v tl /\ EVERY (TRP P) tl
```

If `Node v tl` is a tree, and `P` a predicate over node labels and tree lists, then the *tree recursive predicate* `TRP P` holds of `Node v tl` if and only if `P v tl` and every subtree in `tl` recursively satisfies `TRP P`.

The significance of the predicate is the role it plays in a very general, pre-proved theorem of the HOL system:

```
TY_DEF_THM |- !REP:('b -> 'a ltree) ABS:('a ltree -> 'b)
               P:('a -> ('a ltree list -> bool)).
                (!a. ABS (REP a) = a) /\ (!r. TRP P r = REP (ABS r) = r) ==>
                (!f.
                  ?!fn.
                    !v tl.
                      P v (MAP REP tl) ==>
                        (fn (ABS (Node v (MAP REP tl))) = f (MAP fn tl) v tl))
```

Informally, the theorem says that:

$$\vdash\ !\ P. \cdots \langle :'\,b \textit{ is isomorphic to } TRP\ \ P\rangle ==> \langle \textit{ primitive recursion theorem for } :'\,b\rangle$$

The type variable `:'b` stands for an abstract type isomorphic to the subset of `:'a ltree` characterised by the predicate `TRP P`. The isomorphism is demonstrated by the provision of representation and abstraction functions `REP` and `ABS` satisfying;

```
|- !a:'b. ABS (REP a) = a  /\ !r:('a ltree). TRP P r = (REP (ABS r) = r)
```

Given such an isomorphism, one can obtain, by Modus Ponens, the theorem:

```
PRIM_REC |- !f:('c list) -> 'a -> ('b list) -> 'c.
             ?!fn:('b -> 'c).
               !v:'a  tl:('b list).
```

```
P v (MAP REP tl) ==>
  (fn (ABS (Node v (MAP REP tl))) = f (MAP fn tl) v tl)
```

It is informative to examine precisely what this theorem means. Consider the antecedent of the implication `P v (MAP REP tl)`. Since `tl` is a list of abstract values of type `:'b`, `MAP REP tl` projects every value in `tl` onto its representation. Each such representation will satisfy the characteristic predicate, since `REP` is *onto* the subset described by `TRP P`. Hence `EVERY (TRP P) (MAP REP tl)` must hold. So if the antecedent is true, we have:

```
P v (MAP REP tl) /\ EVERY (TRP P) (MAP REP tl)
```

which, by the definition of `TRP`, is equivalent to:

```
TRP P (Node v (MAP REP tl))
```

But then `Node v (MAP REP tl)` is the correct representation of *some* abstract value `b:'b` denoted by `ABS(Node v (MAP REP tl))`. The implication states that *if* `Node v (MAP REP tl)` is a valid representation of some `b:'b`, *then* the value of the function application `fn b` is given by the application of the non-recursive function `f` applied to the recursive evaluation by `fn` of the subexpressions in `tl`, the representation's label `v` and the values of the subexpressions themselves.

Why is the implication needed? Even though the abstraction function `ABS` is one-to-one on the restricted domain described by `TRP P`, it must be a *total* function. In general,$|:'b| \leq |:'a|$, so it may be the case that more than one tree is mapped to the same abstract value `b:'b`. Consider two such trees, denoted by `Node v' (MAP REP tl')` and `Node v'' (MAP REP tl'')`. Since they are distinct, either $v' \neq v''$ or $tl' \neq tl''$. Suppose the condition `P v (MAP REP tl)` were omitted from the implication. The value of `fn b` is expressed in terms of `v` and `tl`. In the absence of condition `P v (MAP REP tl)`, both pairs `v'`, `tl'` and `v''`, `tl''` are allowed to determine the value of `fn b`. Because the pairs are distinct, `fn b` would not longer denote a unique value, contradicting the requirement that `fn` is a function. By restricting the equality to hold only for correct representations, which are always unique, the inconsistency is avoided.

The complete primitive recursion theorem for `:'b` further states the `fn` is *uniquely* defined by the provision of a case function `f`.

Chapter 2

An Example - Defining the Mutually Recursive Types foo and bar

Before proceeding to the task of automation, it is instructive to consider how a simple set of types might be defined by hand. Consider the types specified by:

$$
\begin{aligned}
foo &= FOO\ num \mid FOOBAR\ foo\ bar \\
bar &= BAR\ bool\ foo
\end{aligned}
$$

2.1 Phase I: Choosing a Representation, Defining the Types and Creating the Constructors

Values of both types can be represented by subsets P_{foo} and P_{bar} of the representing type `((num + one) + bool) ltree` . The idea is that a `foo` value constructed by `FOO n` for some natural number `n` will be represented by a tree in P_{foo} with label `INL (INL n)` and no subtrees. Similarly, a value constructed by `FOOBAR f b` for some `foo` value `f` and `bar` value `b`, will be represented by a tree in P_{foo} with label `INL(INR one)` and the two subtrees representing `f` and `b`. Finally, a `bar` value constructed by `BAR b f` for some boolean `b` and `foo` value `f` corresponds to a tree in P_{bar} with label `INR b` and a single subtree representing `f`. An example abstract value and its corresponding representation is shown in Figure 2.1.

One can inductively define the sets P_{foo} and P_{bar} by requiring them to be the smallest

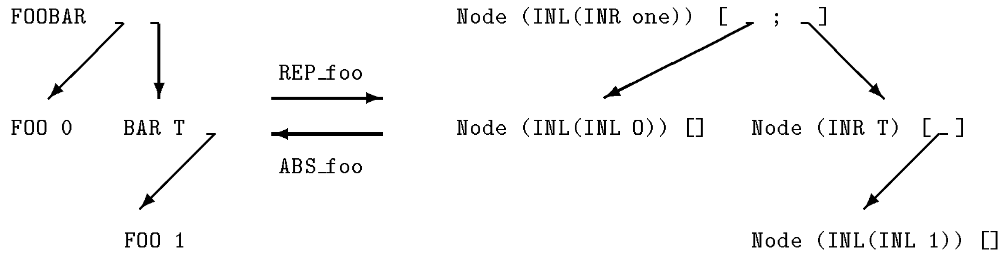

Figure 2.1: The representation of `FOOBAR (FOO 0) (BAR T (FOO 1))`.

sets constructed from the rules:

$$\frac{}{Node\ \ (INL\ \ (INL\ \ n))\ \ tl \in P_{foo}}$$

$$\frac{f \in P_{foo} \quad b \in P_{bar}}{Node\ \ (INL\ \ (INR\ \ one)\ \ [f;b] \in P_{foo}}$$

$$\frac{f \in P_{foo}}{Node\ \ (INR\ \ b)\ \ [f] \in P_{bar}}$$

Unfortunately, inductive definitions of this sort are not supported directly by HOL. Instead, one can carve out the subsets P_{foo} and P_{bar} from the type `:((num + one )+ bool)) ltree` by defining a function `classify_ltree_Exp` which maps all values in P_{foo} and P_{bar} to `1` and `2` respectively, and all others to `0`. Suppose `classify_Exp:((num + one) + bool) -> (num list -> num)` is a function which, given a node's label and the classification of its subtrees, returns the classification of the entire tree. The primitive recursion theorem for labelled trees,

```
ltree_Axiom |- !f. ?!fn. !v tl. fn (Node v tl) = f (MAP fn tl) v tl
```

can be used to define `classify_ltree_Exp` as follows:

1. Specialise the case function `f` to `\nl. \v. \tl. classify_Exp v nl`, to obtain, on β-reduction:

   ```
   |- ?!fn. !v tl. fn (Node v tl) = classify_Exp v (MAP fn tl)
   ```

2. From the unique existence of the function `fn` derive simple existence:

```
|- ?fn. !v tl. fn (Node v tl) = classify_Exp v (MAP fn tl)
```

3. By conservative extension, introduce the new constant `classify_ltree_Exp` satisfying:

```
|- !v tl. classify_ltree_Exp (Node v tl) =
                classify_Exp v (MAP classify_ltree_Exp tl)
```

What form should the definition of `classify_Exp` itself take? The choice is motivated by two rather different uses of the definition. In the following context, we will simply want to apply `classify_ltree_Exp` to a particular tree to determine its class. Later however, we shall encounter the need to derive information about the structure of an *arbitrary* tree, given its classification. One possibility is to use the conditional construct of HOL, where each condition tests both the structure of the label and the classes of the subtrees:

```
|- !v l.  classify v l =
       (  ((?n. v = INL(INL n)) /\ (l = []) )   => 1
        | ( (v = INL (INR one)) /\ (l = [1;2]) ) => 1
        | ( (?b. v = INR b) /\ (l = [1]) ) => 2
        | 0 )
```

However, the nesting of conditional expressions coupled with the explicit testing of the structure of the label `v` makes this intuitive definition difficult to manipulate. An alternative, more appropriate to this application, uses the `||`-operator of Section 1.2.2:

```
classify_Exp |- !v l.
                  classify_Exp v l =
                    ((\FOO_label. (l = []) => 1 | 0) ||
                     (\FOOBAR_label. (l = [1; 2]) => 1 | 0)) ||
                     (\BAR_label. (l = [1]) => 2 | 0) v
```

Given a particular label `v`, `classify_Exp v l` simplifies algebraically to a simple (i. e. non-nested) HOL conditional expression, in which the condition is expressed solely in terms of the subtree classification list `l`, without reference to `v`.

With the help of these functions the characteristic predicates, `is_foo` and `is_bar` of the representing subsets can be defined as follows[1]:

[1]In the actual implementation, the abstractions themselves are used as the characteristic predicates; here, the constants `is_foo` and `is_bar` are defined for ease of exposition.

```
|- is_foo = (\lt. classify_ltree_Exp lt = 1)
|- is_bar = (\lt. classify_ltree_Exp lt = 2)
```

Since the HOL logic requires types to be non-empty, there is an obligation to prove that `is_foo` and `is_bar` hold for at least one tree, or *witness*, each.

```
|- ?x. is_foo x
|- ?x. is_bar x
```

The first follows easily by proving that the classification of `Node (INL n) []`, i.e. the representation of `FOO n` for some arbitrary `n`, is `1`:

```
classify_ltree_Exp (Node (INL (INL n)) [])
```

...expanding by the definition of `classify_ltree_Exp`

```
    = classify_Exp (INL (INL n)) (MAP classify_ltree_Exp [])
```

...since `|- !f. MAP f [] = []`

```
    = classify_Exp (INL (INL n)) []
```

...expanding by the definition of `classify_Exp`

```
    = ((\FOO_label. ([] = []) => 1 | 0) ||
       (\FOOBAR_label. ([] = [1; 2]) => 1 | 0)) ||
       (\BAR_label. ([] = [1]) => 2 | 0)
       (INL (INL n))
```

...simplifying twice by the algebraic properties of `||`

```
    = (\FOO_label. ([] = []) => 1 | 0) ||
      (\FOOBAR_label. ([] = [1; 2]) => 1 | 0)
      (INL n)
    = (\FOO_label. ([] = []) => 1 | 0) n
```

...performing a β-reduction to ignore the value stored in the label ,then simplifying the conditional, finally yields

```
    = (([] = []) => 1 | 0) = 1
```

Since we provided a tree with classification `1` we have shown that $P_foo \neq \emptyset$ or `|- ?x. is_foo x`.

The second theorem is slightly harder to derive. The difficulty arises from the fact that bar has no non-recursive constructors. The witness to P_{bar} needs to be constructed in terms of a subtree in P_{foo}, so here we can use `Node (INR (INR b)) [Node (INL n) []]` , where the subtree is the witness of the preceding discussion. The general case of defining n mutually recursive types requires the construction of a dependency graph relating the types to be defined. A cycle in this graph will indicate the impossibility of producing witnesses for all the types, rendering the type specification invalid.

Having identified an instance of the tree type with inhabited subsets P_{foo} and P_{bar} capable of representing our desired types, the current theory's syntax of types can be extended by the constants `:foo` and `:bar`, along with the assertion of two type definition axioms. Following the discussion in Section 1.3.4 we can introduce the logical constants

```
ABS_foo:  ((num + one) + bool) ltree -> foo
REP_foo:  foo -> ((num + one) + bool) ltree

ABS_bar:  ((num + one) + bool) ltree -> bar
REP_bar:  bar -> ((num + one) + bool) ltree
```

to denote the corresponding abstraction and representation functions satisfying:

```
REP_foo_DEF |- !x' x''. (REP_foo x' = REP_foo x'') ==> (x' = x'') /\
               !x. is_foo x = (?x'. x = REP_foo x')
ABS_foo_DEF |- !a. ABS_foo (REP_foo a) = a /\
               !r. is_foo r = (REP_foo (ABS_foo r) = r)

REP_bar_DEF |- !x' x''. (REP_bar x' = REP_bar x'') ==> (x' = x'') /\
               !x. is_bar x = (?x'. x = REP_bar x')
ABS_bar_DEF |- !a. ABS_bar (REP_bar a) = a /\
               !r. is_bar r = REP_bar (ABS_bar r) = r
```

The result is summarised in Figure 2.2.

Given the abstraction and representation functions, the constructors may be added to the theory as new constants:

```
FOO: num -> foo
FOO_DEF |- !n. FOO n = ABS_foo(Node(INL n)[])

FOOBAR: foo -> bar -> foo
```

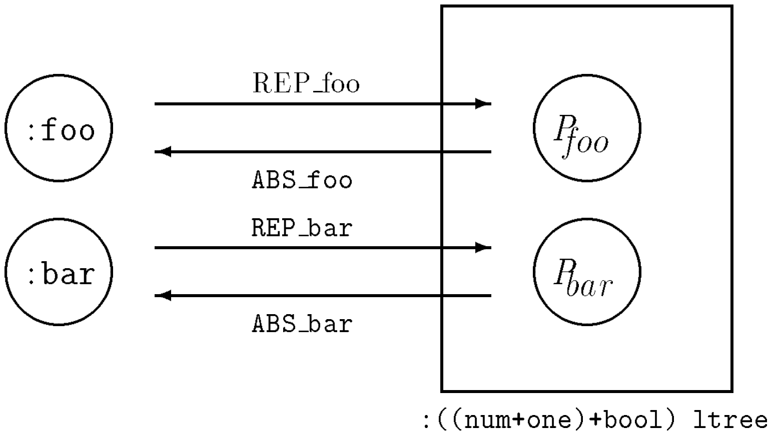

Figure 2.2: Defining the types `:foo` and `:bar`.

```
FOOBAR_DEF |- !f b. FOOBAR f b =
                       ABS_foo(Node(INR(INL one))[REP_foo f;REP_bar b])

BAR: bool -> foo -> bar

BAR_DEF |- !b f. BAR b f = ABS_bar(Node(INR(INR b))[REP_foo f])
```

2.2 Phase II - Deriving a Primitive Recursion Theorem

Now that `:foo` and `:bar` have been defined, we turn towards deriving an abstract axiomatisation of the types. Following Section 1.4, the concise form of this axiomatisation is the primitive recursion theorem:

```
ABS_AX |- !f_FOO f_FOOBAR f_BAR.
           ?!fn:(foo + bar) -> ('foo_range + 'bar_range)
             !n. (OUTL o fn o INL) (FOO n) = f_FOO n  /\
             !f b.
               (OUTL o fn o INL) (FOOBAR f b) =
               f_FOOBAR (OUTL o fn o INL f) (OUTR o fn o INR b) f b  /\
             !b f.
               (OUTR o fn o INR) (BAR b f) = f_BAR (OUTL o fn o INL f) b f
```

With quite a lot of work, mostly in the form of careful algebraic manipulation, it can be obtained from a suitable instance of `TY_DEF_THM` (c. f. Section 1.6).

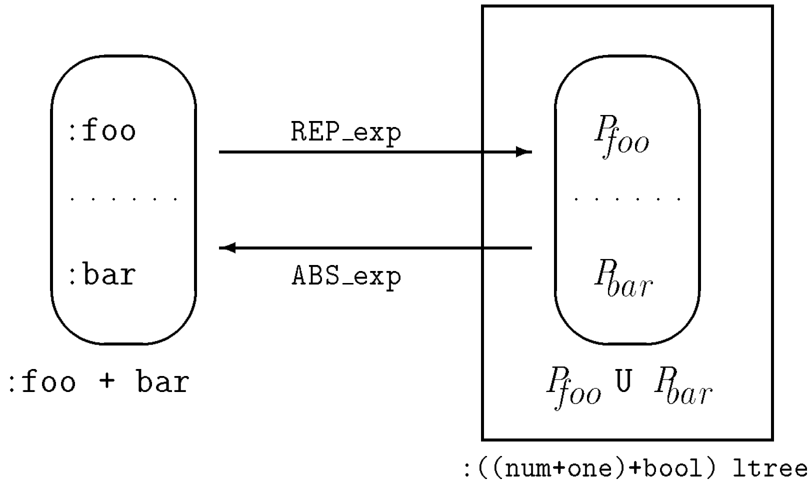

Figure 2.3:

2.2.1 The Derivation of a Basic Primitive Recursion Theorem

`TY_DEF_THM` applies to any abstract types represented by a certain kind of subset of labelled trees. In particular, the representing subset must be selected by the tree recursive predicate `TRP P` for some predicate over node labels and subtree lists `P`.

Unfortunately, neither P_{foo} nor P_{bar} admit characterisations of this form. It is easy to see why: Consider the set P_{foo}. It is described by characteristic predicate `is_foo`. Suppose there existed a predicate `P:((num+one)+bool)->((num+one)+bool ltree) list->bool` such that `TRP P` also characterised P_{foo}, i. e.:

```
is_foo = TRP P
```

Expanding by the definition of `TRP` yields:

```
is_foo = P v tl /\ EVERY (TRP P) tl
```

Substituting by the assumed equality one obtains:

```
is_foo = P v tl /\ EVERY is_foo tl
```

But this is a contradiction since the representation of `FOOBAR f b` satisfies `is_foo` yet has a subtree representing `b:bar` which clearly does not. The equality fails because the subtrees of a given tree are not constrained to satisfy the same characteristic predicate as the entire tree.

Instead, consider the sum type `:foo+bar`, the disjoint union of `:foo` and `:bar`. Since the set of values labeling trees in P_{foo} is disjoint from that of P_{bar}, P_{foo} and P_{bar} are themselves disjoint. By the isomorphism between `:foo` and P_{foo} on the one hand, and `:bar` and P_{bar} on the other, coupled with the fact that $P_{foo} \cap P_{bar} = \emptyset$, `:foo+bar` is itself isomorphic to $P_{foo} \cup P_{bar}$. Thus we can regard `:foo+bar` as an abstract type represented by $P_{foo} \cup P_{bar}$ (c. f. Figure 2.3). This representing subset of labelled trees can be characterised by the predicate `\r. is_foo r \/ is_bar r`. But can this predicate be expressed as `TRP P` for some `P`? The requirement is stated as:

```
(\r. is_foo r \/ is_bar r) = TRP P        (*)
```

By extensionality, the functions are equal if and only if their values agree on all possible trees. So consider an arbitrary tree `Node v tl`. We require:

```
(\r. is_foo r \/ is_bar r) (Node v tl) = TRP P (Node v tl)
```

which, by β-reduction and a single rewriting by the definition of `TRP` is equivalent to:

```
is_foo (Node v tl) \/ is_bar (Node v tl) = P v tl /\ EVERY (TRP P) tl
```

Suppose we 'guess' the solution `is_foo_or_bar` for `P`, where we define `is_foo_or_bar` to be:

```
|- !v tl. is_foo_or_bar v tl =
           is_foo(Node v tl) \/ is_bar(Node v tl)
```

Substituting the guess into the previous equation yields:

```
is_foo (Node v tl) \/ is_bar (Node v tl) =
      is_foo (Node v tl) \/ is_bar (Node v tl) /\ EVERY (TRP is_foo_or_bar) tl
```

Using the equivalence `(A = A /\ B) = (A ==> B)` this simplifies to the implication:

```
is_foo (Node v tl) \/ is_bar (Node v tl) ==>
    EVERY (TRP is_foo_or_bar) tl
```

and by the conjectured equality `(*)`:

```
is_foo (Node v tl) \/ is_bar (Node v tl) ==>
    EVERY (\r. is_foo r \/ is_bar r) tl
```

Intuitively, the implication is clearly true for any tree `Node v tl`, since if `Node v tl` is the correct representation of some `:foo` or `:bar` value, then so are all of its subtrees. A formal proof proceeds by induction on labelled trees. For now, we assume the equality:

```
CHAR_PRED_EQ |- (\r. is_foo r \/ is_bar r) = TRP is_foo_or_bar
```

The bijection between `:foo+bar` and $P_{foo} \cup P_{bar}$ can be demonstrated by defining the representation function `REP`,

```
REP: (foo+bar) -> ((num+one)+bool) ltree
|- REP = REP_foo || REP_bar
```

with the properties:

```
REP_INL |- REP (INL f) = REP_foo f
REP_INR |- REP (INR b) = REP_bar b
```

It is possible to show that `REP` is one-to-one and onto the subset described by:

```
\r. is_foo r \/ is_bar r
```

The proof relies on the corresponding properties of `REP_foo` and `REP_bar` and the disjointness of P_{foo} and P_{bar}.

Assuming that we have established

```
|- TYPE_DEFINITION (\r. is_foo r \/ is_bar r) REP
```

then by the equality `CHAR_PRED_EQ` we derive:

```
|- TYPE_DEFINITION (TRP is_foo_or_bar) REP
```

Using the theorem `ABS_EXISTS_THM` of Section 1.3.4, we can define an abstraction function `ABS` for trees in $P_{foo} \cup P_{bar}$ inversely related to `REP` by:

```
ABS_REP_INVERSE |- !a. ABS (REP a) = a /\
                   !r. (TRP is_foo_or_bar) r = (REP (ABS r) = r)
```

Specialising `TY_DEF_THM` of Section 1.4 to our particular choice of `REP`,`ABS` and the predicate `is_foo_or_bar`; then using Modus Ponens on `ABS_REP_INVERSE` provides us with a primitive recursion theorem on the type `:foo+bar`:

```
SUM_PRIM_REC
|- !f.
    ?!fn: (foo+bar) -> 'c.
      !v tl.
       is_foo_or_bar v (MAP REP tl) ==>
             (fn (ABS (Node v (MAP REP tl))) = f (MAP fn tl) v tl)
```

It can be coerced into the desired primitive recursion theorem for the types foo and bar by:

- Reformulating the theorem in terms of the constructors FOO, FOOBAR and BAR by analysing the constraints imposed on the structure of v and tl by the antecedent of the implication.
- Specialising the case function f to a function which does case analysis on the label v and applies the free variable functions f_FOO, f_FOOBAR and BAR accordingly.
- Generalising over the free functions to obtain the case functions of ABS_AX

The derivation is deferred to Chapter 3 to avoid burdening the reader with unmotivated detail.

2.3 Phase III - Using the Primitive Recursion Theorem

The primitive recursion theorem ABS_AX forms the basis for deriving a host of further theorems characterising the types :foo and :bar.

2.3.1 Defining Functions Over :foo and :bar

The most direct application of ABS_AX is in the actual definition of functions over the new types. Note that the primitve recursion theorem asserts the unique existence of a single function with the sum domain :foo+bar. In general, however, we will be interested in defining functions over the individual components of the sum, say foo_fn:foo->'foo_range and bar_fn:bar->'bar_range. Recalling that the mechanism of constant specification requires only a theorem stating the simple, not necessarily unique, existence of a value satisfying the constant specification, we make use of a theorem facilitating the direct application of constant specification:

```
EXISTS_THM
  |- !f_FOO f_FOOBAR f_BAR.
       ?foo_fn bar_fn.
         (!n. foo_fn (FOO n) = f_FOO n) /\
         (!f b. foo_fn (FOOBAR f b) = f_FOOBAR (foo_fn f) (bar_fn b) f b) /\
         (!b f. bar_fn (BAR b f) = f_BAR (foo_fn f) b f)
```

The derivation from ABS_AX is straightforward:

1. We first specialize the case functions by the free variables `f_FOO`,`f_FOOBAR` and `f_BAR` (variables free in a HOL theorem are treated as universally quantified) and then derive simple existence from unique existence:

```
|- ?fn.
     !n. (OUTL o fn o INL) (FOO n) = f_FOO n  /\
     !f b.
         (OUTL o fn o INL) (FOOBAR f b) =
         f_FOOBAR (OUTL o fn o INL f) (OUTR o fn o INR b) f b  /\
     !b f.
         (OUTR o fn o INR) (BAR b f) = f_BAR (OUTL o fn o INL f) b f
```

2. Using `SELECT_THM` of Section 1.3.1 we can replace the existentially quantified `fn` by the selection `(@fn. ...)` which denotes an arbitrary function satisfying the body of the existential formula.

```
|- (!n. OUTL o (@fn. ...) o INL (FOO n) = f_FOO n) /\
   (!f b.
     OUTL o (@fn. ...) o INL (FOOBAR f b) =
      f_FOOBAR (OUTL o (@fn. ...) o INL f) (OUTR o (@fn. ...) o INR b) f b) /\
   (!b f.
     OUTR o (@fn. ...) o INR (BAR b f) = f_BAR (OUTL o (@fn. ...) o INL f) b f)
```

3. The terms `OUTL o (@fn. ...) o INL` and `OUTR o (@fn. ...) o INR` are functions with the respective types `:foo->'foo_range` and `:bar->'bar_range`. They may thus be used as witnesses to infer:

```
|- ?foo_fn bar_fn.
     (!n. foo_fn (FOO n) = f_FOO n) /\
     (!f b. foo_fn (FOOBAR f b) = f_FOOBAR (foo_fn f) (bar_fn b) f b) /\
     (!b f. bar_fn (BAR b f) = f_BAR (foo_fn f) b f)
```

Generalising over the free variables `f_FOO`,`f_FOOBAR` and `f_BAR` one obtains `EXISTS_THM`.

`EXISTS_THM` is appropriate whenever we wish to simultaneously define two fully specified, possibly mutually recursive functions over `:foo` and `:bar` respectively. In general, however, given a set of mutually recursive types and theorem corresponding to

EXISTS_THM, we will often be interested in defining functions over just a subset of the types. In addition, for a particular type, we may wish to give only a partial specification of the relevant function, i. e. a specification which dictates the functions behaviour over some but not all of the constructors. For the purpose of constant specification, this requires a theorem stating the existence of only those functions we wish to define, quantified over the appropriate subset of the constructor cases given in the original existence theorem.

For instance, consider defining the destructor function DEFOOBAR:foo -> (foo # bar) satisfying the equational specification:

```
DEFOOBAR (FOOBAR f b) = (f,b)
```

Note that result of DEFOOBAR(FOO n) is unspecified and the function does not recurse on either of FOOBAR's arguments. The steps of the definition are as follows:

1. Returning to EXISTS_THM, consider the case function appearing in the second conjunct:

   ```
   f_FOOBAR:'foo_range -> 'bar_range -> foo -> bar -> 'foo_range
   ```

   We can specialise it to the function \fr br f b. (f,b), by instantiating :'foo_range to be the product type :foo#bar. The function ignores its first two arguments, the results of recursive evaluations by foo_fn and bar_fn, and returns the pair of arguments supplied to the FOOBAR constructor. Specialising the irrelevant case functions f_FOO and f_BAR to the free variables f_FOO and f_BAR and performing a simple *beta*-reduction produces:

   ```
   |- ?foo_fn bar_fn.
        (!n. foo_fn (FOO n) = f_FOO n) /\
        (!f b. foo_fn (FOOBAR f b) = (f,b)) /\
        (!b f. bar_fn (BAR b f) = f_BAR (foo_fn f) b f)
   ```

2. The only interesting clause in the theorem is the second conjunct. Clearly by assuming the entire conjunct in the body we can derive the truth of the second conjunct on its own, yielding the implicative theorem:

   ```
   |- ((!n. foo_fn (FOO n) = f_FOO n) /\
   ```

```
    (!f b. foo_fn (FOOBAR f b) = (f,b)) /\
    (!b f. bar_fn (BAR b f) = f_BAR (foo_fn f) b f))
   ==> (!f b. foo_fn (FOOBAR f b) = (f,b))
```

3. The (related) HOL inference rules:

$$\frac{\vdash\ P ==> Q}{\vdash\ (?x.\,P)\ ==>\ (?x.\,Q)}\ if\ x\ free\ in\ Q$$

$$\frac{\vdash\ P ==> Q}{\vdash\ (?x.\,P)\ ==>\ Q}\ if\ x\ not\ free\ in\ Q$$

   allow us to quantify over first `bar_fn` and then `foo_fn` on both sides of the implication. The variable `bar_fn` does not appear free in the antecedent and thus vanishes, leaving:

```
|- (?foo_fn bar_fn.
    (!n. foo_fn (FOO n) = f_FOO n) /\
    (!f b. foo_fn (FOOBAR f b) = (f,b)) /\
    (!b f. bar_fn (BAR b f) = f_BAR (foo_fn f) b f))
   ==> ?foo_fn. (!f b. foo_fn (FOOBAR f b) = (f,b))
```

4. Modus Ponens on the theorems of the first and preceding steps yields:

```
|- ?foo_fn. !f b. foo_fn (FOOBAR f b) = (f,b)
```

   which, by definitional extension, enables us to introduce the new constant `DEFOOBAR`:

```
|- !f b. DEFOOBAR (FOOBAR f b) = (f,b)
```

2.3.2 Proving the Constructors Distinct

Intuitively, one expects that values of the *same* type created using *different* constructors are in fact distinct, regardless of the actual arguments to the constructors. For a given type, this can be proved by defining a function which maps values constructed by the i^{th} constructor C_i to the number i. Consider the arbitrary values v_i and v_j, respectively created by constructors C_i and C_j $(i \neq j)$. They will be mapped to the distinct values i and j. Since applying the function to v_i and v_j yields different results $v_i \neq v_j$.

In the current example the type `bar` admits no such theorem since it has only one constructor. However, for `:foo` we can define a function `map:foo->num` to satisfy:

`|- (!n. map (FOO n) = 0) /\ (!f b. map (FOOBAR f b) = 1)),`

From the properties of functions we have `|- !f f'. (f = f') ==> (map f = map f')`, which on rewriting according to the propositional equivalence `(P ==> Q) = (~Q ==>~P)` yields:

```
L |- !f f'. ~(map f = map f') ==> ~(f = f')
```

Clearly `|- ~(0 = 1)`. So for an arbitrary `n:num`, `f:foo` and `b:bar`:

```
|- ~(map (FOO n) = map (FOOBAR f b))
```

By Modus Ponens on an appropriate instance of the lemma `L` we obtain

```
|- ~(FOO n = FOOBAR f b)
and generalising over the free variables, finally:

|- !n f b. ~(FOO n = FOOBAR f b)
```

2.3.3 Proving The Constructors Are One-To-One

Each n-ary constructor `C` ($n > 0$) of a given type should satisfy:

```
|- (C x1 ... xn = C y1 ... yn) = (x1 = y1 /\ ... /\ xn = yn)
```

We give the general strategy for proving one half of the equality, the other direction is easier.

1. The proof proceeds by defining a destructor function `D` such that [2] :

   ```
   |- D(C x1 ... xn) = (x1, ... ,xn)
   ```

2. One now assumes:

   ```
   C x1 ... xn = C y1 ... yn |- C x1 ... xn = C y1 ... yn
   ```

3. Applying the destructor to both sides of the equality in the conclusion yields:

   ```
   C x1 ... xn = C y1 ... yn |- D(C x1 ... xn) = D(C y1 ... yn)
   ```

   which on simplification be the definition of `D` becomes:

[2] In the actual implementation, no new constant is introduced. The existence of such a function is enough to prove the theorem.

```
C x1 ... xn = C y1 ... yn |- (x1, ... ,xn) = (y1, ... ,yn)
```

4. Rewriting repeatedly by the theorem PAIR_EQ of Section 1.2.1 and discharging the assumption yields one half of our theorem:

```
|- C x1 ... xn = C y1 ... yn ==> (x1 = y1) /\ ... /\ (xn = yn)
```

In the example types, the desired theorems are:

```
|- (!n n'. (FOO n = FOO n') = n = n') /\
     (!f b f' b'. (FOOBAR f b = FOOBAR f' b') = (f = f') /\ (b = b'))
|- !b f b' f'. (BAR b f = BAR b' f') = (b = b') /\ (f = f')
```

2.3.4 The Structural Induction Theorem

Concrete mutually recursive types defined in the manner of Section **??** satisfy structural induction theorems, allowing properties of the types to be proven by case analysis on the constructors. For the current example the theorem reads:

```
ST_IND_THM
  |- !P_foo P_bar.
       (!n. P_foo (FOO n)) /\
       (!f b. P_foo f /\ P_bar b ==> P_foo (FOOBAR f b)) /\
       (!f. P_foo f ==> (!b. P_bar (BAR b f))) ==>
       (!x. P_foo x) /\ (!x. P_bar x)
```

The proof merits a detailed analysis and is deferred to Chapter 3.

2.3.5 Deriving a Cases Theorem

A cases theorem for a concrete type states that any value of the type is the result of applying one of the constructors to some argument values. The theorem can be obtained from a structural induction theorem f for the type.

The current example admits the theorems:

```
|- !x:foo. (?n. x = FOO n) \/ (?f b. x = FOOBAR f b)
|- !x:bar. (?b f. x = BAR b f)
```

We outline the derivation of the cases theorem for type :foo. Starting from the induction theorem ST_IND_THM over the two types, we can specialise the predicate P_bar to `(\b. T)`. Terms previously involving `P_bar` the simplify to `T` and vanish, upon which we are left with a theorem stating the validity of structural induction over the single type `:foo`.

```
|- !P_foo.
      (!n. P_foo (FOO n)) /\
      (!f b. P_foo f ==> P_foo (FOOBAR f b)) ==>
      (!x. P_foo x)
```

The proof proceeds by assuming the existence of an arbitrary `x:foo` such that `x` is *not* the result of an application of either constructor `FOO` or `FOOBAR`; and consequently derives a contradiction. We assume an `x:foo` exists such that every value `FOO n` and `FOOBAR f b` satisfies the predicate `(\f.~(x = f))`. Turning to the structural induction theorem we now specialise `P_foo` to `(\f.~(x = f))`. Note that the variable `x` is free and will become the universally quantified `x` in the final theorem. On β-reduction one obtains:

```
|- (!n.~(x = FOO n)) /\ (!f b.~(x = f) ==>~(x = FOOBAR f b)) ==>
    (!x'.~(x = x'))
```

Turning to the consequent of the implication, by reflexivity one can show `|- x = x`. The consequent states that no value `x':foo` is equal to `x` and is clearly false. Thus the antecedent must be false:

```
|- ~((!n. ~(x = FOO n)) /\ (!f b. ~(x = f) ==> ~(x = FOOBAR f b)))
```

Expressing the implication first as a disjunction and then moving the outermost negation into the formulae as far as possible produces the equivalent:

```
L |- (?n. x = FOO n) \/ (?f b. ~(x = f) /\ (x = FOOBAR f b))
```

Clearly,

```
|- (~(x = f) /\ (x = FOOBAR f b)) ==> (x = FOOBAR f b)+
```

and using the inference rules of page 28 to existentially quantify over `f` and `b`:

```
|- (?f b. ~(x = f) /\ (x = FOOBAR f b)) ==> (?f b. x = FOOBAR f b)
```

Together with lemma `L` and generalising over `x` we ultimately infer the cases theorem for `:foo` postulated above.

Chapter 3

Automating It All

The aim of the project is to fully automate the sort of derivations and definitions we examined in 2 for the large class of concrete types described by the general grammar of Section 1.4. Considerable insight into the task can be gained from studying the existing techniques employed in the definition of singly recursive types. Thomas Melham explains the theoretical background in great detail [?]. A rich, additional source of both knowledge and actual code is his existing implementation of a package for defining singly recursive types and deriving their auxiliary theorems.

Though the axiomatisation of higher order logic in the HOL system consists of only a handful of axioms and primitive inference rules, a large number of powerful *derived* rules, expressed as meta-language programs written in terms of the primitives, are available to the user. This is both a boon and a bane: trivial proof steps can be left unspecified by the user of program, but a gain in generality is often accompanied by a loss in efficiency. Crafting efficient task specific inference rules requires considerable experience with the system and is symptomatic of the difference between formal reasoning on paper, and proof *engineering* on a machine. For this reason, the implementation of the mutually recursive type definition facility borrows, where it can, from the less general package, which has benefitted form numerous revisions by an experience HOL 'programmer'.

The primary concern of the implementation is to provide *practical* support for reasoning about mutually recursive types. The axiomatisation of such types naturally divides into three phases:

Phase I The definition of the types and their constructors by conservative extension.

Phase II The derivation of a corresponding primitive recursion theorem or abstract axiom.

Phase III The derivation of auxiliary theorems from the abstract axiom.

The mechanisation of these stages can be split accordingly. Phase II requires the greatest amount of non-trivial inference and its automation a corresponding level of experience with the HOL system. In addition, it was not clear at the outset whether the inferences, though feasible, could be performed at all efficiently enough to be applicable to large type specifications. The structure of the desired primitve recursion theorems is sufficiently regular to allow them to be synthesised and asserted directly as axioms. Whilst such assertions may conceivably introduce inconsistency into the logic, due to their simplicity and the fact that the presuppose the successful completion of Phase I, they can nevertheless be accepted to a large degree of confidence. Even with the existence of a consistency-preserving implementation of Phase II, the availability of a fast but nominally unsafe tool for the same task would be of considerable utility in any application in which frequent revisions to large type specifications are made.

Given the time constraint, it was felt that the benefits of a functional, complete set of tools, that however failed to fully guarantee consistency, outweighed the more purist merits of a logically safe, yet incomplete and possibly impractically inefficient package. In light of these considerations, it was decided that the implementation of a consistency-preserving Phase II be postponed in favour of fully completing Phases I and III, the set of tools most needed in practice. Phases I and III have been successfully implemented. Moreover, a strategy for the *efficient* derivation of the primitive recursion has been identified in sufficient detail to guide an actual implementation.

The following sections give an overview of the main tactics used to automate Phases I and II, and an account of the steps involved in the mechanisation of Phase III.

3.1 Phase I: From an Informal Type Specification to a Formal Type Definition

The main steps of Phase I are to:

1. Parse an informal type specification.

2. Verify that the specification describes a valid set of types.

3. Choose an appropriate tree representation for the collection of types and carve out the appropriate representing subsets.

4. Prove that each representing set is inhabited by at least one value.

5. Perform the definitional type extensions yielding a set of representation and abstraction functions.

6. Employ the abstraction and representation functions to define the constructors of the types as new constants.

The input to the package is an informal type specification in the syntax of Section *mutrecdef*. The existing parser for singly recursive types has been adopted in a slightly modified form. Its purpose is to:

- Collect the set of new type names.

- Associate each new type with a list of its constructors.

- Associate each constructor with a list of its argument types, where each type is marked as existent or recursive (i. e. to be defined).

3.2 Tracing the Type Dependencies

Though type operators may be added to a HOL theory by definitional extension, they must be provenly inhabited by at least one element, or *witness.* Recalling that the set of values we wish to identify with an informal type specification is the set generated by all *finite* applications of the constructors to argument values, it should come as no surprise that some specifications in our syntax do not in fact describe inhabited types. Consider the trivial counter-example of the specification of the impossible type `:loopy`:

$$loopy \quad ::= \quad LOOP\ loopy \tag{3.1}$$

Any value of type `:loopy` must have been constructed by an application of `LOOP` to an argument of type `:loopy`, which in turn must have been constructed by an application of `LOOP` to an argument of type `:loopy` ad infinitum. For a singly recursive type specification, the problem arises whenever the type fails to have a non-recursive constructor.

In the context of mutually recursive specifications, the situation is more difficult to detect correctly. For instance, though the type `:bar` of Chapter 2 fails to have a

---

Input: *A finite set of types* $T = \{T_1, \ldots, T_n\}, n > 0$*;*
For each T_i*, a set of constructors* $\{C_{i1}, \ldots, C_{im_i}\}, m_i > 0$*;*
For each constructor C_{ij} *the set* $R_{ij} \subseteq T$*, of recursive types on which it depends.*

Output: *A total order on the types in* T*, where each* T_i *is paired with one of its constructors* C_{ij} ***or*** *an indication that no such order exists.*

Method:

```
J := T
while there exists an i,j such that
      T_i ∈ J and R_ij ∩ J = ∅ do
      output (T_i, C_ij)
      J := J − {T_i}
if J ≠ ∅ then raise 'failure'
```

---

non-recursive constructor, the type `:foo`, on which it depends, *does*, yielding the finite expression `BAR b (FOO n)`.

Fortunately, whether a set of new types, T, admits a complete set of witnesses can be determined purely syntactically from the specification according to the algorithm in Figure 3.2. We associate each new type, $T_i \in T$, with its constructors and each constructors C_{ij} with the set $R_{ij} \subseteq T$ of recursive types appearing in its specification. The algorithm maintains a set J, where the types in J still require witnesses, and the types in $T - J$ have already been determined to be non-empty. Initially, $J = T$. On each iteration, an attempt is made to produce a witness for some type in J, using the knowledge that witnesses can be produced for all types in $T - J$. Intuitively, $R_{ij} \cap J$, for $T_i \in J$, denotes the set of types still requiring witnesses before constructor C_{ij} can be applied to produce a witness for type T_i. Note that a witness can be constructed for some type $T_i \in J$ if and only if it has a constructor C_{ij} satisfying $R_{ij} \cap J = \emptyset$. Thus on each iteration, J is searched for some T_i with constructor C_{ij} such that $R_{ij} \cap J = \emptyset$. If none is found the algorithm terminates, otherwise the algorithm iterates on the smaller set $J - T_i$.

Termination is guaranteed since on each iteration either a T_i is removed from J or no such T_i exists and the loop guard is made false. The specification is valid if and only if, on termination, there are no types unaccounted for, i.e. $J = \emptyset$. If all types are accounted for, then the order in which types were removed, coupled with the information of which constructor made each removal possible, is sufficient guidance for actually constructing the witnesses and thus the existence proofs required by the type definition facility of HOL.

3.3 Phase One

3.3.1 Choosing A Representation

Values of the recursive types `Aty`,..., `Bty` are represented using various subsets of a common instance of the labelled tree type `:'a ltree`.

The type variable `:'a` must be instantiated to a type allowing each constructor and the values of its non-recursively typed arguments to be stored in a manner which *uniquely* identifies the constructor used. A suitable instance of `:'a` is determined as follows. For each constructor with possibly empty list of non-recursive argument types `:ty1`,...,`tyn`, we create a compound *leaf* type, to record the values of the non-recursive arguments. If $n > 1$ the leaf is a nested product `:(ty1 # ... # tyn)`. If $n = 1$ the leaf is simply `:ty1`; otherwise the constructor takes no non-recursive arguments and is assigned the dummy leaf `:one`. The entire collection of constructors yields a list of leaves, l.

The tree labelling type `:'a` is instantiated to `:s`, the disjoint union or *summation* of the leaves in l. Thus each leaf is associated with a unique injection or *labelling* function, for creating values of `:s`, by the application to an appropriate leaf value. To cater for the efficient treatment of specifications with large numbers of constructors, the actual structure of the sum type is chosen to be a *complete binary tree* of the summands in l. The benefits are two-fold:

- The average size of terms denoting node labels is only $O(log(n))$ rather than the $O(n)$ associated with a linear sum, where n is the total number of constructors.
- Applications of the `||` and `++` operators can be simplified in correspondingly fewer steps.

Here we have a simple example of logically equivalent alternatives lending themselves more or less appropriately to *efficient* proof automation.

Now that we have found an instance, `:s ltree` of `:'a ltree` to collectively represent value of the types `Aty`,..., `Bty` the task remains of identifying the specific disjoint subsets of `:s ltree` isomorphic to the individual types.

The general approach is to associate each new type with a distinct numeral starting from 1 called its *class*. Each value in `:s ltree` is mapped to the corresponding class of the value it represents, and to class 0 if it does not represent any value at all. A tree `Node v tl` represents a value of class `n` if and only if

- The injection function containing the leaf value of `v` is paired with a constructor `C` of the type with class `n`;*and*
- the classifications of the subtrees in `tl` correspond in number and order to the classes of the recursively typed arguments of `C`.

From the primitive recursion theorem for `'a ltree` it is possible to derive a general theorem to facilitate the definition of such recursive classification functions:

```
|- !classify. ?classify_ltree:'a ltree -> num.
      !v tl. classify_ltree (Node v tl) = classify v (MAP classify_ltree tl)
```

The case function `classify:'a -> num list -> num` is meant to take a node's label `v` and the classification of its subtrees,`tl` as a list `l`, to return the class of the node. In the general setting, the function can be expressed as a nested `||` expression, reflecting the structure of the sum type `:s`. When applied to a particular node label `v`, and subtree classification list `l`, the label `v` is repeatedly stripped of its outermost injection function, and the classifier of its outermost `||` operator, until a leaf value `v'` is reached. At this point, all `||`-operators have vanished leaving a leaf *function* of the original `||` expression. Supposing the leaf `v'` captures the non-recursive arguments of a constructor `C`, with class `n`. Moreover, let the classes of `C`'s recursive arguments be constrained to the values `c1,...,cj`, in that order. The resulting term yields the application:

```
(\v'.(l = [c1,...,cj] => n | 0)) v'
```

The actual leaf value `v'` is irrelevant to the classification and vanishes on β-reduction. As desired, the classification is dependent solely on the list of subtree classifications `l`.

This form of definition of `classify` is preferred to the nested conditional expression briefly considered in Section 2.2 for both the efficiency with which simplification can be performed (i. e. algebraic manipulation versus proof) and the *uniform* result of such simplification: a simple conditional term.

With the provision of a case function `classify`, by definitional extension we can introduce a constant `classify_ltree` to denote the required recursive classification function. The subset of `:s ltree` used to represent each type `Xty` can now be described by the characteristic predicate `(\r.classify_ltree r = n)` , where `n`$\neq$ `0`, is the class assigned to `Xty`.

3.3.2 Proving the Representing Sets Non-Empty

It remains to prove that each representing subset is non-empty.

The ordering on the types provided by the algorithm of Figure 3.2 provides a recipe for constructing an element of each subset, i.e. for constructing a tree t such that `classify_ltree t = n`, for each class `n`.

Consider the ordering $(T_1, C_1), \ldots, (T_n, C_n)$. By the nature of its construction, constructor C_i's, $1 \leq i \leq n$, set of recursive argument types R_i will satisfy $R_i \subseteq \{T_1, \ldots, T_{i-1}\}$. In other words, constructor C_i depends on at most the recursive types $T_1, \ldots, T_{i-1}$. These properties of the abstract constructors transfer directly to their representations as labelled trees. Given the existence of witnesses $w_1, \ldots, w_{i-1}$ for the representing subsets of types $T_1, \ldots, T_{i-1}$ respectively, the witness, w_i, for type T_i can be obtained as follows:

1. The constructor C_i has an associated leaf type,`:ty` and injection function `IN` for storing the non-recursive arguments (if any) at the root of its representing tree. We choose a free variable `v` to stand for some set of arbitrary non-recursive arguments. `(IN v)` labels the root of the tree w_i.

2. Suppose C_i takes k recursive arguments Then the witness w_i must be supplied with k subtrees, each drawn from the appropriate subset of labelled trees used to represent the type of the corresponding argument. But every argument type is in $R_i \subseteq \{T_1, \ldots, T_{i-1}\}$. Hence we can us the associated witnesses drawn from $\{w_1, \ldots, w_{i-1}\}$.

Clearly constructor C_1 depends on *no* recursive types and the corresponding witness w_1 possesses an empty list of subtrees. The other witness can be obtained by following the prescribed order.

Proving that the tree w_i is indeed in representing subset of T_i reduces to evaluating `classify_ltree` w_i to establish that `classify_ltree` w_i `= n` where `n`. is the class of T_i. Since the class of a tree does not depend on the value stored at the leaf of its label, but only on the structure of the injection `IN`, the arbitrary choice of `v` above is justified.

The actual implementation is slightly more subtle: it avoids explicitly creating the *entire tree* for each witness. Instead, it chooses the label for witness w_i as before, but uses Hilbert's selection operator to denote subtrees from the appropriate representing sets. Once a subset has been shown to be inhabited, the denoted witness will provably evaluate to its proper class, i. e. satisfy the subset's characteristic predicate.

For instance, we recall from Section 2.1 that the witness for `:bar`,

```
Node (INR b) [Node (INL(INL n)) []]
```

is expressed directly in terms of the witness for `:foo`. Consider the alternative formulation:

```
Node (INR b) [@r. is_foo r]
```

On proving that the witness `Node (INL(INL n)) []` satisfies the predicate `is_foo` we derive the inhabitance theorem:

```
|- ?r. is_foo r
```

By `SELECT_THM` this is equivalent to

```
|- is_foo (@r. is_foo r)
```

and rewriting by the definition of `is_foo` and a single β-reduction:

```
L |- classify_ltree_Exp (@r. is_foo r) = 1
```

Now consider applying `classify_ltree_Exp` to the witness for `:bar`. After a *single* unfolding by the definition of `classify_ltree_Exp` and `classify_Exp` and simple algebraic manipulation we obtain

```
|- classify_ltree_Exp (Node (INR b) [@r. is_foo r]) =
   ([classify_ltree_Exp (@r. is_foo r)] = [1] => 2 | 0)
```

Which simplifies directly to `2` using lemma `L` to rewrite the test of the conditional expression.

So even in the general case, using this approach, each witness can be expressed as a tree of height one, whose classification follows by a single unfolding of the definition of the classification function. The result is a collection of inhabitance theorems of the form

```
 |- ?t. (\r.classify_ltree r = n) t
```

for each representing subset.

3.3.3 Defining The Types And Constructors As Conservative Extensions of The HOL Logic

At this point we have identified a tree type `:s tree` capable of representing values of all the mutually recursive types `Aty`,..., `Bty` in the informal specification. The representing subsets have been described by the provision of characteristic predicates expressed in terms of the classification function. Finally, each predicate has been shown to describe a non-empty set.

Thus the three pre-requisites to actually *defining* the types have been fulfilled and we are free to assert the corresponding *type definition axioms* of Section 1.3.4. The type operators `Aty`,..., `Bty` are consequently added to the HOL logic by conservative extension. Note that because of the single underlying representation, each type operator will be declared as being of the same arity a, where a is the number of free type variables occurring in the *entire* specification.

This is actually rather counter-intuitive. To illustrate the point, consider the informal type specification:

$$\begin{aligned} boo &= BOO\ 'a \\ hoo &= HOO\ 'b \end{aligned} \tag{3.2}$$

The types are conceivably independent, yet the type operators resulting from the definition will both have arity 2, $not 1$. This can lead to subtle conceptual errors in which two values `BOO 1` and `BOO 1` which intuitively appear to be equal are not even comparable because of their differing types `:(num,num)boo` and `(num,bool)boo`.

Each type definition axiom states the existence of a one-to-one and onto representation function from the abstract type to its representing subset.

Because the bijective properties of the representation functions are explicitly needed in the derivation of the primitive recursion theorem (for instance, we need the bijective properties of both `REP_foo` and `REP_bar` to derive that the composite function `REP` is one-to-one and onto the union of the representing sets, $P_{foo} \cup P_{bar}$), they are defined directly from the type definition axiom by constant specification. The corresponding abstraction functions are specified to be the inverses of the newly introduced representation functions using the theorem `ABS_EXISTS_THM` of Section 1.3.4. The constructors are defined to map their arguments onto the appropriate abstraction of their representation as labelled trees.

3.4 Phase Two: Deriving the Primitive Recursion Theorem

In this section we first generalise the steps of Section 2.2.1 to the derivation of a basic primitive recursion theorem for an arbitrary number of types. We then return to the example to complete the derivation of the desired abstract axiom `ABS_AX` from the point at which we left off in Section 2.2.1: the basic primitive recursion theorem `SUM_PRIM_REC`. The derivation is presented in great detail because the strategy it employs can easily be generalised and adopted as the basis of an *efficient* tool for the automatic derivation of primitive recursion theorems.

3.4.1 Deriving a Basic Primitive Recursion Theorem

`TY_DEF_THM` of Section 1.6 forms the basis for the derivation of basic primitive recursion theorems. Its application is restricted to types which are isomorphic to subsets of labelled trees characterised by predicates of the form `TRP P`. A tree satisfies the predicate `TRP P` if and only if the label and subtree list satisfy `P` and every subtree recursively satisfies `TRP P`. Thus `P` may be seen as a constraint on the label and subtree list and `TRP` as a function mapping this constraint onto every node in the entire tree. Thus every subtree rooted within the tree must satisfy the same predicate `TRP P`. `TY_DEF_THM` originates from the existing package supporting singly recursive type definitions. Any subtree of the representation of a value of a singly recursive type clearly represents some value of that very same type. Hence it comes as no surprise that singly recursive types always admit representations described by `TRP P`, for some `P`. Given a label of a node, the predicate `P` merely needs to constrain the *number* of the node's subtrees to correspond to the number of recursive arguments to the associated constructor. Similarly, the failure of mutually recursive types to admit characteristic predicates of this form follows from the fact that a representing tree may have a subtree representing a value of a *different* type.

Returning to the general case of the types `:Aty`,...,`:Bty`, represented by subsets of `:s ltree` with the respective characteristic predicates `is_A`,...,`is_B`. The representing subsets of the types are mutually disjoint, since the classification function maps any two trees in different subsets to distinct classes. Since each type is isomorphic to its representing type, the disjoint sum `:(Aty ...` + Bty)+ is isomorphic to the union of all the disjoint representing sets. The union is characterised by the predicate `is_rep` defined as the disjunctive predicate:

```
|- is_rep = \r. (is_A r) \/ ... \/ (is_B r)
```

Letting REP_A, ..., REP_B denote the concrete types' corresponding representation functions, the isomorphism can be demonstrated by defining the composite representation function REP mapping from the sum of the types to the union of their representations as follows:

```
|- REP = REP_A || ... || REP_B
```

and then proving that it is in fact bijective on the union of the representing sets. The proof follows from the bijective nature of each REP_A, ..., REP_B *and* the disjointness of the representing sets. Using ABS_EXISTS_THM of Section 1.3.4, one can then introduce the related abstraction function ABS mapping from trees to the disjoint sum such that:

```
ABS_REP |- !a. ABS (REP a) = a /\
           !r. is_rep r = (REP (ABS r) = r)
```

Clearly any subtree of a tree serving to represent an abstract value also represents an abstract value of a possibly different type. So any tree satisfying is_rep has the property that all of its subtrees satisfy is_rep as well. If we define the predicate on node labels and subtree lists P as:

```
  |- P = \v tl. is_A (Node v tl) \/ ... \/ is_B (Node v tl)+
```

then it is possible to show by induction on trees that:

```
  |- is_rep = TRP P
```

from which we obtain:

```
  |- !a. ABS (REP a) = a /\
     !r. (TRP P) r = (REP (ABS r) = r)
```

Specialising TY_DEF_THM by REP, ABS and P, then using Modus Ponens on the previous theorem yields the basic abstract axiom:

```
  |- !f.
      ?!fn.
       !v tl.
         P v (MAP REP tl) ==>
           (fn (ABS (Node v (MAP REP tl))) = f (MAP fn tl) v tl)
```

Rewriting by definition of P the theorem becomes more informative:

```
|- !f.
    ?!fn.
     !v tl.
        is_A (Node v (MAP REP tl)) \/
               ...                 \/
        is_B (Node v (MAP REP tl)) ==>
           (fn (ABS (Node v (MAP REP tl))) = f (MAP fn tl) v tl)
```

Each disjunct in the antecedent places constraints on the structure of v and tl. The following section explains how, by making theses constraints explicit, the theorem could be rephrased in terms of the constructors of the types, without reference to their representation.

3.4.2 Reformulating the Basic Primitive Recursion Theorem in Terms of the Constructors

Returning to the example of Chapter 2, the theorem SUM_PRIM_REC of Section 2.2.1 can be coerced into the desired primitive recursion theorem displayed at the beginning of Section 2.2. Identifying an efficient strategy for performing this transformation is crucial if we are to provide HOL with a *practical* and *consistency preserving* package for defining large-scale mutually recursive types. The following exposition is intended to be detailed enough to guide an actual implementation.

We start with the body of the theorem SUM_PRIM_REC.

```
!v tl.
   is_foo_or_bar v (MAP REP tl) ==>
   (fn (ABS (Node v (MAP REP tl))) = f (MAP fn tl) v tl)
```

Each of the following steps results in an equivalent expression for the body.

1. By case analysis on the sum structure of the label v, the body can be split into a conjunction of three clauses. Each clause corresponds to a single constructor and gives rise to a distinct clause of ABS_AX.

```
(!n tl.
   is_foo_or_bar (INL (INL n)) (MAP REP tl) ==>
```

```
    (fn (ABS (Node (INL (INL n)) (MAP REP tl))) =
    f (MAP fn tl) (INL (INL n)) tl)) /\
  (!o' tl.
    is_foo_or_bar (INL (INR o')) (MAP REP tl) ==>
    (fn (ABS (Node (INL (INR o')) (MAP REP tl))) =
    f (MAP fn tl) (INL (INR o')) tl)) /\
  (!b tl.
    is_foo_or_bar (INR b) (MAP REP tl) ==>
    (fn (ABS (Node (INR b) (MAP REP tl))) =
    f (MAP fn tl) (INR b) tl))
```

We shall only present the transformation of the third conjunct concerning the `BAR` constructor; the other conjuncts are dealt with similarly. Note that the universally quantified boolean `b` will become the first argument to the `BAR` constructor.

2. Replacing `is_foo_or_bar` by its definition yields a disjunction in the antecedent:

```
  !b tl.
    is_foo (Node (INR b) (MAP REP tl)) \/
    is_bar (Node (INR b) (MAP REP tl)) ==>
    (fn (ABS (Node (INR b) (MAP REP tl))) =
    f (MAP fn tl) (INR b) tl)
```

Intuitively, the first disjunct should simplify to `F` and vanish, since no value of type `:foo` has a representation with label `INR b`. Expressing `is_foo` and `is_bar` in terms of the classification function yields:

```
  !b tl.
    (classify_ltree_Exp (Node (INR b) (MAP REP tl)) = 1) \/
    (classify_ltree_Exp (Node (INR b) (MAP REP tl)) = 2) ==>
    (fn (ABS (Node (INR b) (MAP REP tl))) =
    f (MAP fn tl) (INR b) tl)
```

We now expand `classify_ltree_Exp` by its definition in terms of `classify_Exp`. In turn, `classify_Exp` can be replaced by the nested `||`-expression, which on algebraic simplification according to `sigma_DEF` results in:

```
!b tl.
   (((MAP classify_ltree_Exp (MAP REP tl) = [1]) => 2 | 0) = 1) \/
   (((MAP classify_ltree_Exp (MAP REP tl) = [1]) => 2 | 0) =
   2) ==>
   (fn (ABS (Node (INR b) (MAP REP tl))) =
   f (MAP fn tl) (INR b) tl) :thm
```

Consider the first disjunct. Since the conditional expression on the left hand side of the equation can only evaluate to either `2` or `0`, the equality must be false and the disjunct vanishes. On the other hand, the second disjunct is satisfied whenever the conditional expression evaluates to `2`. This occurs precisely when the condition `MAP classify_ltree_Exp (MAP REP tl) = [1]` is true and we obtain:

```
!b tl.
   (MAP classify_ltree_Exp (MAP REP tl) = [1]) ==>
   (fn (ABS (Node (INR b) (MAP REP tl))) =
   f (MAP fn tl) (INR b) tl)
```

And on rewriting by `|- MAP f (MAP g l) = MAP (f o g) l`:

```
!b tl.
   (MAP (classify_ltree_Exp o REP) tl = [1]) ==>
   (fn (ABS (Node (INR b) (MAP REP tl))) =
   f (MAP fn tl) (INR b) tl)
```

3. The equality in the antecedent imposes a constraint on the structure of the list `tl:(foo+bar) list`. The constraint is expressed by the general theorem over lists `l` of unit length:

```
|- (MAP f l = [y]) = (?x. (l = [x]) /\ (f x = y))
```

By application of this theorem, the clause rewrites to:

```
!b tl.
   (?x. (tl = [x]) /\ (classify_ltree_Exp o REP x = 1)) ==>
   (fn (ABS (Node (INR b) (MAP REP tl))) =
   f (MAP fn tl) (INR b) tl)
```

4. The conjunct `classify_ltree_Exp o REP x = 1` merely states that `REP x` $\in P_{foo}$ and is equivalent to the term `is_foo (REP x)`. Now `REP_foo` is surjective on P_{foo} which is expressed by the theorem `|- !x. is_foo x = (?f:foo. x = REP_foo f)`. Performing the substitutions transforms our clause into:

```
!b tl.
   (?x. (tl = [x]) /\ (?f. REP x = REP_foo f)) ==>
   (fn (ABS (Node (INR b) (MAP REP tl))) =
   f (MAP fn tl) (INR b) tl)
```

   The conjunct bears further simplification. The composite representation function `REP` can be shown to satisfy:

```
REP_ONE_ONE |- !x' x''. (REP x' = REP x'') = (x' = x'')
```

   By subsequent applications of `REP_INL` of Section 2.2.1 and `REP_ONE_ONE`, `?f. REP x = REP_foo f` simplifies to `?f. x = INL f`:

```
!b tl.
   (?x. (tl = [x]) /\ (?f. x = INL f)) ==>
   (fn (ABS (Node (INR b) (MAP REP tl))) =
   f (MAP fn tl) (INR b) tl)
```

5. The antecedent of the implication constrains the consequent to hold only for certain values of `tl`. By rephrasing the entire formula directly in terms of these values, the implication can be removed. Using the equivalences

```
|- ((?x. P) ==> Q) = (!x.(P ==> Q)), where x is not free in Q
|- ((P /\ Q) ==> R) = (P ==> (Q ==> R))
```

   The existential quantification of `x` within the antecedent can be replaced by a universal quantification over the entire implication; The conjunction then gives way to a nested implication leaving:

```
!b tl x.
   (tl = [x]) ==>
   (?f. x = INL f) ==>
   (fn (ABS (Node (INR b) (MAP REP tl))) =
   f (MAP fn tl) (INR b) tl)
```

Resorting to a generalisation of the equivalence `|- (!x. x = tm ==> P[x]) = P[tm/x]` allows us to dispense with `tl` and the outermost implication to yield.

```
!b x.
  (?f. x = INL f) ==>
  (fn (ABS (Node (INR b) (MAP REP [x]))) =
  f (MAP fn [x]) (INR b) [x])
```

The mappings in the consequent can now be unfolded. A similar sequence of steps leads to the replacement of occurrences of `x` by `INL f'` (note the renaming to `f'` to avoid confusion with the case function, `f`, appearing previously in the consequent):

```
!b f'.
  fn (ABS (Node (INR b) [REP (INL f')])) =
  f [fn (INL f')] (INR b) [INL f']
```

6. The last step is to express the left hand side of the equation in terms of the constructor `BAR`. By the theorem `REP_INL` the term `REP (INL f')` rewrites to `REP_foo f'`. From the definition of the constructor `BAR` and the inverse relationship of `REP_bar` and `ABS_bar`, it is fairly straightforward to show:

   ```
   |- REP_bar (BAR b f) = Node (INR b) [REP_foo f]
   ```

   and consequently:

   ```
   !b f'.
     fn (ABS (REP_bar (BAR b f'))) =
     f [fn (INL f')] (INR b) [INL f']
   ```

   The theorem `REP_INR` of Section 2.2.1 together with the inverse relationship of `ABS` and `REP`, finally produces a term involving no references to representations.

   ```
   |- !b f'. fn (INR (BAR b f')) = f [fn (INL f')] (INR b) [INL f']
   ```

Assuming all clauses of the body of Step 1. have been re-expressed in terms of the constructors, after replacing some compound applications by function composition and substituting the final term into the body of `SUM_PRIM_REC` we obtain the theorem:

```
|- !f.
     ?!fn.
       (!n. fn o INL (FOO n) = f [] (INL (INL n)) []) /\
       (!f' b.
         fn o INL (FOOBAR f' b) =
         f [fn o INL f'; fn o INR b] (INL (INR one)) [INL f'; INR b]) /\
       (!b f'.
         fn o INR (BAR b f') = f [fn o INL f'] (INR b) [INL f'])
```

The case function `f:'c list -> ((num+one)+bool) -> (foo+bar) list -> 'c` takes three arguments: a list of evaluations by `fn` of the recursively typed subexpressions, a node label, and a list of the subexpressions themselves. The structure of the label uniquely determines the form and number of values appearing in the list arguments. We can specialise `f` to a function which does case analysis on the label and applies the generic functions `f_FOO`, `f_FOOBAR` and `f_BAR` to the appropriately extracted components of its arguments. In addition, the specialised function must allow values of types `:foo` and `:bar` to be mapped to the possibly distinct ranges `:'foo_range` and `:'bar_range`. This is achieved by instantiating the variable type `:'c` to the sum `:'foo_range+'bar_range`, and arranging that the results of `f_FOO`, `f_FOOBAR` and `f_BAR` applications are injected accordingly. The appropriate function is expressed in terms of the `||` operator, the list destructors `HD` and `TL` and the sum projection functions `OUTL` and `OUTR`:

```
\rl v l.
    ((\n. INL (f_FOO n)) ||
     (\o'. INL (f_FOOBAR (OUTL (HD rl)) (OUTR (HD (TL rl)) (OUTL (HD l))
                         (OUTR (HD (TL l)))))
    ) ||
    (\b. INR (f_BAR (OUTL (HD rl)) b (OUTL (HD l)))) v
```

On substitution into the previous theorem and considerable yet uninteresting algebraic simplication, we obtain the new theorem:

```
|- ?!fn.
     (!n. fn o INL (FOO n) = INL (f_FOO n))                        /\
     (!f' b.
       fn o INL (FOOBAR f' b) =
```

```
    INL (f_FOOBAR (OUTL (fn o INL f')) (OUTR (fn o INR b))
                  (OUTL (INL f')) (OUTR (INR b))))          /\
  (!b f'.
    fn o INR (BAR b f') =
    INR (f_BAR (OUTL (fn o INL f')) b (OUTL (INL f'))))
```

Noting that the right hand side of each equality is an injection into a sum type, the injections can be removed by applying the appropriate projection function to both sides of each equality. The combinations of injections and projections on the right hand sides cancel out by theorems `OUTL` and `OUTR`. Rewriting multiple function applications as function compositions and generalising over the newly introduced, generic case functions `f_FOO`, `f_FOOBAR` and `f_BAR` finally yields the desired form of the primitive recursion theorem, `ABS_AX`, promised at the start of Section 2.2.

3.5 Phase Three: Making it Useful

Though a primitive recursion theorem provides a concise axiomatization of a set of mutually recursive types, it is of *little* direct use in everyday theorem proving. Of far greater utility are the theorems stating:

- The validity of structural induction.
- That every value of a recursive type must have been created by one of its constructors (a 'cases' theorem)
- The constructors of recursive types are one-to-one
- The constructors of a recursive type yield distinct values.

Intuitively, because instances of the last three theorem families concern only a individual concrete types, it might be expected that their derivation is exactly the same as for singly recursive types. This is indeed the case, allowing the existing support in the form of MelHam's package for singly recursive types to be successfully applied. The strategies for generating these proofs are relatively simple and are adequately described in Section 2.3. For this reason, this section is devoted mainly to the derivation of the structural induction theorem. We deliver the proof of `ST_IND_THM` promised in Section 2.3. It should be clear to the reader how the proof generalises from 2 to an arbitrary number of types.

3.5.1 The Derivation of a Structural Induction Theorem

The derivation of the structural induction theorem `ST_IND_THM` utilises the property that primitive recursive definitions specify *unique* functions, and thus departs directly from the abstract axiomatisation `ABS_AX`. It is helpful to first dissect the formal definition of unique existence:

```
UNIQUE_DEF |- (?!x. P x) = (?x. P x) /\ (!x y. P x /\ P y => (x = y))
```

Whilst the first conjunct requires the mere existence of a value `x` satisfying the predicate `P`, the second goes further to state that any two values satisfying `P` must, in fact, be equivalent.

The details of the derivation are important in that they are somewhat more involved than those of proving a structural induction theorem for a singly recursive type. A broad outline follows:

1. Since we will be reasoning about predicates, we instantiate the variable types `'foo_range` and `:'bar_range` of `ABS_AX` to `:bool`. For the moment, we specialise the case functions to the free variables `f_FOO`, `f_FOOBAR` and `f_BAR` leaving behind the body quantified by the unique existence of `fn:(foo+bar)->(bool+bool)`. From this, essentially using the second conjunct of the definition of unique existence, one can deduce:

```
|- !x y.
      ((!n. OUTL o x o INL (FOO n) = f_FOO n) /\
      (!f b.
        OUTL o x o INL (FOOBAR f b) =
        f_FOOBAR (OUTL o x o INL f) (OUTR o x o INR b) f b) /\
      (!b f.
        OUTR o x o INR (BAR b f) = f_BAR (OUTL o x o INL f) b f)) /\
      (!n. OUTL o y o INL (FOO n) = f_FOO n) /\
      (!f b.
        OUTL o y o INL (FOOBAR f b) =
        f_FOOBAR (OUTL o y o INL f) (OUTR o y o INR b) f b) /\
      (!b f.
        OUTR o y o INR (BAR b f) = f_BAR (OUTL o y o INL f) b f) ==>
      (x = y)
```

which merely states that any two functions satisfying the primitive recursive definition are equal.

2. Consider the composite functions `(\f. T) ++ (\b. T)` and `P_foo ++ P_bar` which rely on the `++`-operator of Section 1.2.2. Both have type `:(foo+bar)->(bool+bool)`. Whilst `(\f. T):foo->bool` and `(\b. T):bar->bool` are predicates which hold true of *any* argument, `P_foo` and `P_bar` stand for arbitrary predicates.

   As an aside, it is instructive to reflect on an actual application of the structural induction theorem `ST_IND_THM` in the proof of some specific properties `Q_foo` and `Q_bar`. Presumably, the goal will be to demonstrate that `Q_foo` and `Q_bar` hold for *all* values of `:foo` and `:bar`. But this is equivalent to showing that `Q_foo = (\f. T)` and `Q_bar = (\b. T)`.

   Returning to our proof, if we can show that:

```
((\f. T) ++ (\b. T)) = (P_foo ++ P_bar)
```

   then, due to the characteristics of `++`, we will in fact have proven `!f. P_foo f` and `!b. P_bar b`. Using the theorem of the previous step, with `x` and `y` specialised to `(\f. T) ++ (\b. T)` and `P_foo ++ P_bar` respectively, we will only need to show that `(\f. T) ++ (\b. T)` and `P_foo ++ P_bar` satisfy a common primitive recursive definition. The definition is given by appropriate specialisations of the currently free case functions `f_FOO`, `f_FOOBAR` and `f_BAR`.

   For now, we substitute our composite functions into the preceding theorem. From the definition of `++` it is possible to prove the equalities:

```
|- OUTL o (P_foo ++ P_bar) o INL = P_foo
|- OUTR o (P_foo ++ P_bar) o INR = P_bar
|- OUTL o ((\f. T) ++ (\b. T)) o INL = (\f. T)
|- OUTR o ((\f. T) ++ (\b. T)) o INR = (\b. T)
```

   enabling us to simplify the intermediate theorem firstly to:

```
|- ((!n. (\f. T) (FOO n) = f_FOO n) /\
   (!f b.
     (\f. T) (FOOBAR f b) = f_FOOBAR ((\f. T) f) ((\b. T) b) f b) /\
   (!b f. (\b. T) (BAR b f) = f_BAR ((\f. T) f) b f)) /\
```

```
(!n. P_foo (FOO n) = f_FOO n) /\
(!f b. P_foo (FOOBAR f b) = f_FOOBAR (P_foo f) (P_bar b) f b) /\
(!b f. P_bar (BAR b f) = f_BAR (P_foo f) b f) ==>
((\f. T) ++ (\b. T) = P_foo ++ P_bar)
```

and on subsequent β-reduction and generalisation over the case functions:

```
|- !f_FOO f_FOOBAR f_BAR.
     ((!n. T = f_FOO n) /\
     (!f b. T = f_FOOBAR T T f b) /\
     (!b f. T = f_BAR T b f)) /\
     (!n. P_foo (FOO n) = f_FOO n) /\
     (!f b.
       P_foo (FOOBAR f b) = f_FOOBAR (P_foo f) (P_bar b) f b) /\
     (!b f. P_bar (BAR b f) = f_BAR (P_foo f) b f) ==>
     ((\f. T) ++ (\b. T) = P_foo ++ P_bar)
```

3. Ignoring the first three conjuncts of the antecedent for the present, we examine how suitable choices for the case functions, which are really just predicates, may transform the remaining conjuncts into the antecedent of `ST_IND_THM`. The base case of the induction theorem emerges by specialising `f_FOO` to `(\n. T)`. Focusing on the step case due to the recursive constructor `BAR`, it seems that the truth of `P_bar(BAR b f)` should at least be allowed to depend on the inductive hypotheses `P_foo f`. On the other hand, `P_bar (BAR b f)` may hold directly, independently of `P_foo f`. Specialising `f_BAR` to `(\Pf b f. Pf \/ P_bar(BAR b f))` expresses these considerations. Similar reasoning leads to a choice of

   ```
   (\Pf Pb f b. (Pf /\ Pb) \/ P_foo(FOOBAR f b))
   ```

   for the case function `f_FOOBAR`. Performing the substitutions and several β-reductions results in:

   ```
   |- ((!n. T = T) /\
      (!f b. T = T /\ T \/ P_foo (FOOBAR f b)) /\
      (!b f. T = T \/ P_bar (BAR b f))) /\
      (!n. P_foo (FOO n) = T) /\
   ```

```
(!f b.
  P_foo (FOOBAR f b) =
  P_foo f /\ P_bar b \/ P_foo (FOOBAR f b)) /\
(!b f. P_bar (BAR b f) = P_foo f \/ P_bar (BAR b f)) ==>
((\f. T) ++ (\b. T) = P_foo ++ P_bar)
```

The first three conjuncts of the antecedent simplify to `T`. Using the propositional equivalence `(P = (Q \/ P)) = (Q => P)` on the step cases produces the desired implications:

```
|- (!n. P_foo (FOO n)) /\
   (!f b. P_foo f /\ P_bar b ==> P_foo (FOOBAR f b)) /\
   (!b f. P_foo f ==> P_bar (BAR b f)) ==>
   ((\f. T) ++ (\b. T) = P_foo ++ P_bar)
```

4. The final step is to rewrite the consequent of the implication. By extensionality, the two function are equal if and only if they are point-wise equal for all values of type `:foo+bar`, giving the equivalent expression:

```
!s:foo+bar. ((\f. T) ++ (\b. T)) s = (P_foo ++ P_bar) s
```

Case analysis on the structure of the sum `s` and algebraic simplification of the composite function application provides us with:

```
!x. INL T = INL (P_foo x) /\ !x. INR T = INR (P_bar x)
```

which by the one-one-ness of `INL` and `INR` yields the expected:

```
!x. P_foo x /\ !x. P_bar x
```

A final generalisation over the free predicate `P_foo` and `P_bar` produces `ST_IND_THM`.

Note that starting form the primitive recursion theorem in the manner above, we have inferred the most general induction theorem allowing simultaneous induction over two types. In the general case of a set of n mutually recursive types we will often be interested in performing an induction over just a subset of the types. By specialising the undesired predicates of the induction theorem to predicates which are always satisfied,i. e. `\x.T`, on simplification, an induction theorem over the domain types of the remaining predicates is obtained. In the limiting case where all but one of the predicates is specialised to `\x.T` we obtain a structural induction theorem for a single type, allowing the application of the existing package for proving cases theorems.

3.5.2 Automating the Definition of Mutually Recursive Functions

When the primitive recursion theorem for a set of types is obtained, the implementation derives from it a theorem stating the (non-unique) existence of functions, one for each type, defined by primitive recursion. These *existence* theorems are sufficient for defining functions over the types. An example of such a theorem and its derivation is given as `EXISTS_THM` of Section 2.3.1.

A tool has been provided to allow functions to be defined automatically from informal equational specifications. Given an existence theorem and an equational specification, the package performs extensive pattern matching to produce the correct case functions with which to specialise the existence theorem. The package caters for:

- Sets of functions over arbitrary subsets of the types.
- Partial functions that are undefined for some constructor values.
- Curried functions in which the recursion may be on any one of the arguments.
- Equational specifications in which the clauses may appear in an arbitrary order, independently of the structure of the existence theorem.

The package defines constants denoting the functions and a theorem resembling the equational specification. The equations of the theorem are used as rewrites in the evaluation of function applications.

Chapter 4

Conclusion

4.1 The State of the Implementation

Phases I and III are fully implemented and, on their own, are ensured to preserve consistency. All of the general theorems discussed in the report can be derived automatically for arbitrary valid type specifications. The primitive recursion theorem produced by Phase II is the only 'theorem' which is neither derived nor a conservative extension but asserted as axiom. The discussion heading Chapter 3 justifies this concession to possible inconsistency. The goal of producing a *practical* system has been achieved.

4.2 Further Work

The only obstacle to the implementation of a *consistency-preserving* version of Phase II, the derivation of the primitive recursion theorem, is time. The steps in the derivation are clear and an inefficient version should be fairly easy to code. The inefficiency would stem not from the choice of a poor strategy, but rather from an unoptimised, naive mechanisation of that strategy. Furthermore, room is open for the development of tools to support goal directed reasoning, such as a structural induction tactic, which, given a goal to prove, produces the corresponding step and base cases as subgoals. At the moment, the type specification parser, though functional, is completely independent of the HOL type parser. A re-implementation of the parser using ML yacc and the existing HOL type parser would provide a more elegant interface.

4.3 Concluding Remarks

We have shown that it is possible to define and axiomatise a large class of mutually recursive types in a manner admitting efficient mechanisation. The result has practical value, since here is no other existing *usable* package for the same task, yet *is* the demand for such a tool exists. It is felt that it would be worthwhile to carry on with the implementation of a consistency-preserving automation of Phase II, with the ultimate aim of adding the package to the standard HOL library.

Acknowledgements

I would like to thank my supervisor Kevin Mitchell for the weekly chats and encouragement. I would also like to thank Tom Melham for sharing his views with me a day before the hand-in, and Kevin again for prodding me to go share them.

Bibliography

[Melham89] T.F. 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.