A Theory of Lists for HOL
Based on Higher-Order Functions

Paul Curzon, University of Cambridge &
Wai Wong, Hong Kong Baptist College

Abstract

We describe the new HOL theory of lists. A notable feature is that it is based largely on higher-order functions: 'fold left' and 'fold right'. By using general theorems about these functions we have been able to provide tools which can quickly prove theorems about both system and user-defined list constants avoiding the need of induction.

1 Introduction

Lists are one of the fundamental data types of computer science. Due to their flexibility they occur in a wide range of applications. As such it is important that a theorem-proving system contains a comprehensive theory of lists with support tools for reasoning about them. Previously the HOL system contained only an ad-hoc theory of lists with few tools. We describe recent efforts to remedy this. We overview the list theory in the most recent release of HOL, and describe our attempts to provide general-purpose tools for reasoning about lists. These tools make use of the fact that many inductively defined constants can also be defined uniformly in terms of the higher-order functions 'fold right' and 'fold left'. By exploiting theorems and tools for these functions, a wide range of general theorems can quickly be proven.

2 The New List Theory of HOL

The list theory in the core system of HOL88 version 2.02 and which will appear in HOL90 version 7 has undergone a major overhaul from earlier versions. Previously support for lists in HOL was in two parts: (i) a small *ad hoc* collection of definitions and theorems with few tools in the core system, and (ii) the `more_lists` library which contained contributions from many users. This was incoherent and inadequate for many applications. The new theory is entirely in the core HOL system. The `more_lists` library has been deleted, though many of its definitions and theorems are included in the new theory. (Old code is kept in `contrib`). All definitions and theorems previously in the core HOL list theory have been retained.

The aim of the new theory is to provide a coherent, structured and flexible framework for reasoning about lists. Much of the theory is based on the work of Bird and Wadler [1]. The theory contains a comprehensive set of 44 constant definitions and approximately 300 theorems about them. The revision was influenced by the following factors:

- to keep change to the old core theory to a minimum so as to cause as little inconvenience to users as possible;
- to develop a very small set of fundamental functions as the base of the theory;
- to define other functions in terms of these fundamental functions.

We will describe the most important constants in the remainder of the text of this paper. A list of all constant definitions, together with some notes on the naming convention of the theorems we followed can be found in the appendices.

2.1 List constructors

In the original theory, the basic constructor CONS was provided for adding elements onto the front of a list. SNOC is defined to mirror CONS, allowing an element to be added to the tail end of a list. It was one of the most widely used features from the `more_lists` library. It is used extensively in functional programming circles, but no consensus appears to have been reached about its name. We therefore used the name from the `more_lists` library for consistency.

Suppose that in some applications, a list of boolean values is used to represent a natural number with the most significant bit at the head of the list, e.g., $[\mathsf{T};\mathsf{F};\mathsf{T};\mathsf{T}]$ denotes 11. One may define a constant, say NVAL, which converts such a list of booleans to a natural number using CONS as follows.

$$\begin{array}{ll}\vdash & (\mathsf{NVAL}\ [\,] = 0)\wedge \\ & (\mathsf{NVAL}(\mathsf{CONS}\ x\ l) = ((\mathsf{BV}\ x) \times (2\ \mathsf{EXP}(\mathsf{LENGTH}\ l))) + (\mathsf{NVAL}\ l))\end{array}$$

where $\mathsf{BV}\ x = (x \Rightarrow 1 \mid 0)$. This definition is both complex and computationally expensive. It can be defined in a much simpler way using SNOC.

$$\begin{array}{ll}\vdash & (\mathsf{NVAL}\ [\,] = 0)\wedge \\ & (\mathsf{NVAL}(\mathsf{SNOC}\ x\ l) = (2 \times (\mathsf{NVAL}\ l)) + (\mathsf{BV}\ x))\end{array} \tag{1}$$

This can be done using `new_recursive_definition` with the induction theorem `SNOC_Axiom` as below.

```
new_recursive_definition false SNOC_Axiom `NVAL`
"(NVAL [] = 0) /\
 (NVAL (SNOC x l) = (2 * (NVAL l)) + (BV x))";;
```

Of course, the order of the elements of the list could have been reversed. However, if the application being modelled naturally used a particular ordering of elements, such a reversal might add extra complexity and make specifications less easy to follow by someone familiar with the original order. Thus, whilst SNOC is not absolutely necessary, it is often convenient.

2.2 Higher-order list functions

Perhaps the most significant new constants are the higher-order fold right and fold left functions FOLDR and FOLDL, respectively. These functions are specified as below:

$$\mathsf{FOLDR}\ f\ e\ [x_0; x_1; \ldots; x_n] = f\ x_0(f\ x_1(\ldots(f\ x_n\ e)\cdots))$$
$$\mathsf{FOLDL}\ f\ e\ [x_0; x_1; \ldots; x_n] = f(\ldots(f(f\ e\ x_0)x_1)\cdots)x_n$$

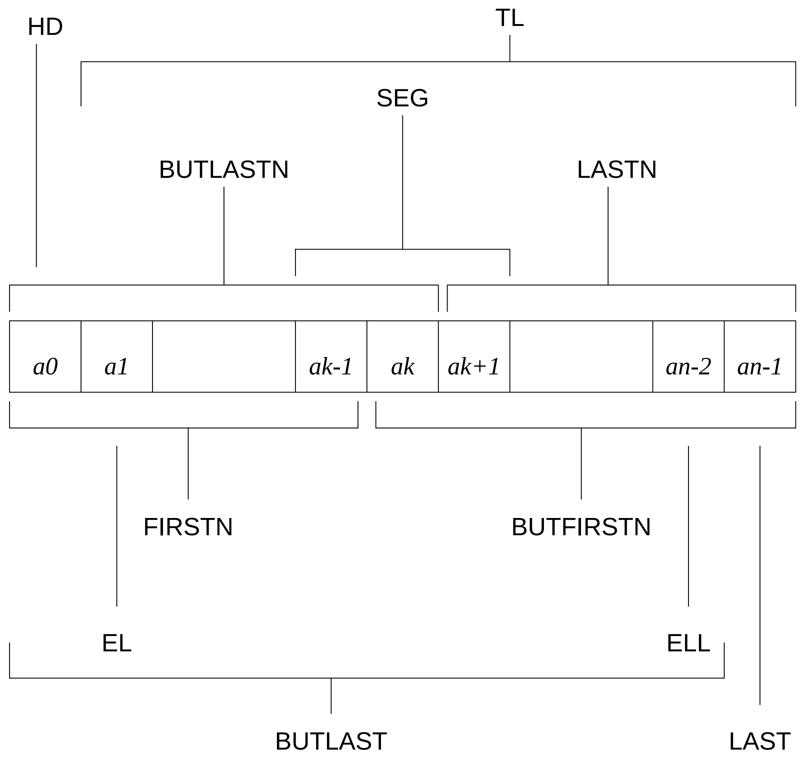

Figure 1: List segment operators

Apart from the segment operations, most list operations can be conveniently defined in terms of either of these two higher-order functions. For example, the constant NVAL may be defined in terms of FOLDL:

$$\vdash \ \mathsf{NVAL}\ l = \mathsf{FOLDL}\ (\lambda e\ x.(2 \times e) + x)\ 0\ l$$

These higher-order functions enable us to provide tools for automatically producing a wide range of theorems about both system and user-defined constants. More on this will be described in Section 3.

2.3 List segments

We provide a comprehensive range of operators for accessing segments of lists as shown in Figure 1. The most basic function is SEG which returns a segment of m elements of a given list starting from the kth element as specified below.

$$\mathsf{SEG}\, m\, k\, [x_0; x_1; \ldots; x_k; \ldots; x_n] = [x_k; x_{k+1}; \ldots; k_{k+m-1}]$$

The index of elements starts from 0. Theorems relating each of the segment constants to SEG are provided.

2.4 Induction tactics

The tools provided in the new theory fall into three categories:

- induction tactics,
- conversions for deriving more general list theorems, and
- evaluation conversions.

The latter two kinds of tools are discussed in more detail in Sections 3 and 4, respectively. We very briefly describe the induction tactics here.

Previously, the only induction tactic for lists was `LIST_INDUCT_TAC` which performs structural induction over a list from the front of the list. This has been retained. The new induction tactic, `SNOC_INDUCT_TAC` can be used to do induction from the tail end of a list. It is useful for proving theorems defined recursively using SNOC. For example, properties of NVAL, defined earlier, can be proved using induction from the end of the list. A call to `SNOC_INDUCT_TAC` on a goal of the following form

```
!l. ...(NVAL l)  ...
```

would generate the familiar base case and a step case using SNOC, that is, a goal of the form ...`(NVAL (SNOC x l))`... with an inductive assumption ...`(NVAL l)`.... This goal could be rewritten with the recursive definition 1.

`EQ_LENGTH_INDUCT_TAC` and `EQ_LENGTH_SNOC_INDUCT_TAC` perform structural induction on a pair of lists whose lengths are identical using CONS and SNOC, respectively, as the constructor.

3 A Tool for Proving Theorems about Lists

3.1 The fold functions

The fold functions provide a template for inductive definitions over lists. For example, consider the definition of FOLDR[1].

$$\vdash \begin{array}{l}(\forall f\ e.\, \mathsf{FOLDR}\ f\ e\ [\,] = e)\wedge \\ (\forall f\ e\ x\ l.\, \mathsf{FOLDR}\ f\ e\ (\mathsf{CONS}\ x\ l) = f\ x\ (\mathsf{FOLDR}\ f\ e\ l))\end{array} \tag{2}$$

The *base element* (e) gives the result of an inductive definition in the base case. The *step function* (f) is the action to be performed in the step case on the current element and the result of the recursive call. Definitions in terms of FOLDR can thus be used in place of recursive definitions. For example, in the old list theory, APPEND was recursively defined as follows.

$$\vdash \begin{array}{l}(\forall l.\, \mathsf{APPEND}\ [\,]\ l = l)\wedge \\ (\forall l_1\ l_2\ x.\, \mathsf{APPEND}\ (\mathsf{CONS}\ x\ l_1)\ l_2 = \mathsf{CONS}\ x\ (\mathsf{APPEND}\ l_1\ l_2))\end{array} \tag{3}$$

It could be defined equally well in terms of FOLDR as:

$$\vdash \forall l_1\ l_2.\, \mathsf{APPEND}\ l_1\ l_2 = \mathsf{FOLDR}\ \mathsf{CONS}\ l_2\ l_1 \tag{4}$$

From this definition, the two conjuncts in the recursive definition (Equation 3) can be derived easily. We first specialise l_1 in Equation 4 with either [ ] or a CONS

[1] We will restrict our discussion to FOLDR, though FOLDL is treated similarly.

$\vdash\ \forall l_1\, l_2.\, \mathsf{APPEND}\ l_1\ l_2 = \mathsf{FOLDR\ CONS}\ l_2\ l_1$
$\vdash\ \forall f\, l.\, \mathsf{MAP}\ f\ l = \mathsf{FOLDR}\ (\lambda\, x\, l'.\ \mathsf{CONS}(f\ x)\ l')\ []\ l$
$\vdash\ \forall l.\, \mathsf{SUM}\ l = \mathsf{FOLDR}\ \$+\ 0\ l$
$\vdash\ \forall l.\, \mathsf{FLAT}\ l = \mathsf{FOLDR\ APPEND}\ []\ l$
$\vdash\ \forall l.\, \mathsf{LENGTH}\ l = \mathsf{FOLDR}\ (\lambda\, x\, n.\ \mathsf{SUC}\ n)\ 0\ l$
$\vdash\ \forall l.\, \mathsf{NULL}\ l = \mathsf{FOLDR}\ (\lambda\, x\, l'.\ \mathsf{F})\ \mathsf{T}\ l$
$\vdash\ \forall P\, l.\, \mathsf{FILTER}\ P\ l = \mathsf{FOLDR}\ (\lambda\, x\, l'.\ (P\ x) \Rightarrow (\mathsf{CONS}\ x\ l') \mid l')\ []\ l$
$\vdash\ \forall l.\, \mathsf{REVERSE}\ l = \mathsf{FOLDR\ SNOC}\ []\ l$
$\vdash\ \forall P\, l.\, \mathsf{ALL\_EL}\ P\ l = \mathsf{FOLDR}\ (\lambda\, x\, l'.\ P\ x \wedge l')\ \mathsf{T}\ l$
$\vdash\ \forall P\, l.\, \mathsf{SOME\_EL}\ P\ l = \mathsf{FOLDR}\ (\lambda\, x\, l'.\ P\ x \vee l')\ \mathsf{F}\ l$
$\vdash\ \forall y\, l.\, \mathsf{IS\_EL}\ y\ l = \mathsf{FOLDR}\ (\lambda\, x\, l'.\ (y = x) \vee l')\ \mathsf{F}\ l$
$\vdash\ \forall P\, l.\, \mathsf{PREFIX}\ P\ l = \mathsf{FOLDR}\ (\lambda\, x\, l'.\ (P\ x) \Rightarrow (\mathsf{CONS}\ x\ l') \mid [])\ []\ l$
$\vdash\ \forall x\, l.\, \mathsf{SNOC}\ x\ l = \mathsf{FOLDR\ CONS}\ [x]\ l$
$\vdash\ \forall\, l.\, \mathsf{AND\_EL}\ l = \mathsf{FOLDR}\ \$\wedge\ \mathsf{T}\ l$
$\vdash\ \forall\, l.\, \mathsf{OR\_EL}\ l = \mathsf{FOLDR}\ \$\vee\ \mathsf{F}\ l$
$\vdash\ \forall l.\, (\mathsf{FLAT} \circ \mathsf{REVERSE})\ l = \mathsf{FOLDR}\ (\lambda\, l_1\, l_2.\ \mathsf{APPEND}\ l_2\ l_1)\ []\ l$

Figure 2: Definitions using FOLDR

expression. For example, specialising with a CONS expression (and eliminating the universal quantifier on l_2) gives the theorem:

$$\vdash\ \mathsf{APPEND}\ (\mathsf{CONS}\ x\ l_1)\ l_2 = \mathsf{FOLDR\ CONS}\ l_2\ (\mathsf{CONS}\ x\ l_1)$$

This can then be rewritten with the recursive definition of FOLDR (Equation 2) to derive the following theorem.

$$\vdash\ \mathsf{APPEND}\ (\mathsf{CONS}\ x\ l_1)\ l_2 = \mathsf{CONS}\ x\ (\mathsf{FOLDR\ CONS}\ l_2\ l_1)$$

Finally, the definition (Equation 4) can be used in reverse and the resulting equations generalised over the free variables to give the required theorem.

$$\vdash\ \forall x\, l_1\, l_2.\, \mathsf{APPEND}\ (\mathsf{CONS}\ x\ l_1)\ l_2 = \mathsf{CONS}\ x\ (\mathsf{APPEND}\ l_1\ l_2)$$

This process is relatively easy to automate and is quick because it involves only specialisation and simple rewriting. Essentially, we simply take a general theorem about the application of FOLDR to CONS, and specialise the function f with an appropriate operation to give a theorem about a different constant applied to CONS. In fact, a proof procedure implementing this process has been written using the new theory. It will be described later in this section.

In order to provide a wider base with which the automatic proof procedure can work, it would have been desirable to define the list functions in terms of the fold functions (as in Equation 4) and to derive the inductive form from it. This was not done for consistency with earlier versions of HOL. Nevertheless, equational theorems giving equivalent expression in terms of the fold functions are included in the theory. A list of such theorems that we have derived is given in Figure 2.

3.2 Monoids

Given theorems about the application of FOLDR to constants other than CONS, we can obtain a wide range of theorems in a similar process to that outlined above. Bird and Wadler [1] describe some suitable theorems. For example they give the following theorem about APPEND.

$$\begin{array}{ll} \vdash & \forall f\ e.\,\mathsf{MONOID}\ f\ e \supset \\ & \quad (\forall l_1\ l_2.\,\mathsf{FOLDR}\ f\ e\ (\mathsf{APPEND}\ l_1\ l_2) = f\ (\mathsf{FOLDR}\ f\ e\ l_1)\ (\mathsf{FOLDR}\ f\ e\ l_2)) \end{array} \tag{5}$$

To use this theorem in the way outlined above requires that the base element and step function, e and f, form a monoid. That is f is associative and e is both its left and right identity. The base elements and step functions for many of the common constants listed in Figure 2 have this property. For example, the base element 0 and the step function $+$ for SUM form a monoid, i.e., $\vdash$ MONOID $\$+\ 0$. With this fact, we can specialise and rewrite (5) to arrive at the following theorem.

$$\vdash\ \ \mathsf{SUM}\ (\mathsf{APPEND}\ l_1\ l_2) = (\mathsf{SUM}\ l_1) + (\mathsf{SUM}\ l_2)$$

Similar theorems about ALL_EL and FLAT, for example, as well as many other constants can be obtained in this way.

In a first order logic, these theorems would be proved by a series of very similar inductive proofs. The higher order theorems, such as 2 and 5 can be thought of as a proof plan for these theorems. Thus, instead of performing many similar proofs for different constants, a single inductive proof about the higher-order function is sufficient. A much simpler, non-inductive proof is used to derive specific theorems for each constant.

3.3 Function commutativity

Whilst Theorem 5 is sufficient for proving many useful theorems, there are common constants such as LENGTH and NULL for which the step function and base element do not form a monoid. We have identified the more general theorem given below which can be used to prove a much wider set of theorems. Theorem 5 is a special case of this theorem in which f and g are identical.

$$\begin{array}{ll} \vdash & \forall g\ f.\,\mathsf{FCOMM}\ g\ f \supset \\ & \quad (\forall e.\,\mathsf{LEFT\_ID}\ g\ e \supset \\ & \quad\quad (\forall l_1\ l_2.\,\mathsf{FOLDR}\ f\ e\ (\mathsf{APPEND}\ l_1\ l_2) = \\ & \quad\quad\quad\quad g\ (\mathsf{FOLDR}\ f\ e\ l_1)\ (\mathsf{FOLDR}\ f\ e\ l_2))) \end{array} \tag{6}$$

It relies on a form of commutativity for binary functions, FCOMM, which is a generalised version of associativity.

$$\vdash\ \ \forall f\ g.\,\mathsf{FCOMM}\ f\ g = (\forall x\ y\ z.\, g\ x\ (f\ y\ z) = f\ (g\ x\ y)\ z)$$

We can derive Theorem 7 below about LENGTH by specialising g and f in Theorem 6 with $+$ and $(\lambda x\ n.\ \mathsf{SUC}\ n)$, respectively.

$$\vdash\ \ \mathsf{LENGTH}\ (\mathsf{APPEND}\ l_1\ l_2) = (\mathsf{LENGTH}\ l_1) + (\mathsf{LENGTH}\ l_2) \tag{7}$$

Thus, given theorems about the fold functions, many useful theorems about lists can be derived using essentially the same proof procedure. A conversion which encodes this procedure has been developed and is the subject of the next sub-section.

$\vdash\ \ \forall f\ e.\,\mathsf{FOLDR}\ f\ e\ [\,] = e$
$\vdash\ \ \forall f\ e\ x\ l.\,\mathsf{FOLDR}\ f\ e\ (\mathsf{CONS}\ x\ l) = f\ x\ (\mathsf{FOLDR}\ f\ e\ l)$
$\vdash\ \ \forall f\ e\ x\ l.\,\mathsf{FOLDR}\ f\ e\ (\mathsf{SNOC}\ x\ l) = \mathsf{FOLDR}\ f (f\ x\ e)\ l$
$\vdash\ \ \forall f\ e\ l_1\ l_2.\,\mathsf{FOLDR}\ f\ e\ (\mathsf{APPEND}\ l_1\ l_2) = \mathsf{FOLDR}\ f (\mathsf{FOLDR}\ f\ e\ l_2)\ l_1$
$\vdash\ \ \forall g\ f.\,\mathsf{FCOMM}\ g\ f \supset (\forall e.\,\mathsf{LEFT\_ID}\ g\ e \supset$
$\quad (\forall l_1\ l_2.\,\mathsf{FOLDR}\ f\ e\ (\mathsf{APPEND}\ l_1\ l_2) = g\ (\mathsf{FOLDR}\ f\ e\ l_1)\ (\mathsf{FOLDR}\ f\ e\ l_2)))$
$\vdash\ \ \forall g\ f.\,\mathsf{FCOMM}\ g\ f \supset (\forall e.\,\mathsf{LEFT\_ID}\ g\ e \supset$
$\quad (\forall l.\,\mathsf{FOLDR}\ f\ e\ (\mathsf{FLAT}\ l) = (\mathsf{FOLDR}\ g\ e (\mathsf{MAP}\ (\mathsf{FOLDR}\ f\ e)\ l)))$
$\vdash\ \ \forall f.\,\mathsf{COMM}\ f \wedge \mathsf{ASSOC}\ f \supset$
$\quad (\forall g\ l.\,\mathsf{FOLDR}(\lambda x\ l.\ f\ (g\ x)\ l)\ e\ (\mathsf{REVERSE}\ l) = (\mathsf{FOLDR}(\lambda x\ l.\ f\ (g\ x)\ l)\ e\ l)$

Figure 3: Theorems about FOLDR

$\vdash\ \ \mathsf{SUM}\ [\,] = 0$
$\vdash\ \ \mathsf{APPEND}\ (\mathsf{CONS}\ x\ l_1)\ l_2 = \mathsf{CONS}\ x\ (\mathsf{APPEND}\ l_1\ l_2)$
$\vdash\ \ \mathsf{APPEND}\ l_1\ (\mathsf{CONS}\ x\ l_2) = \mathsf{APPEND}\ (\mathsf{SNOC}\ x\ l_1)\ l_2$
$\vdash\ \ \mathsf{FLAT}\ (\mathsf{SNOC}\ x\ l) = \mathsf{APPEND}\ (\mathsf{FLAT}\ l)\ x$
$\vdash\ \ \mathsf{LENGTH}\ (\mathsf{APPEND}\ l_1\ l_2) = \mathsf{LENGTH}\ l_1 + \ \mathsf{LENGTH}\ l_2$
$\vdash\ \ \mathsf{NULL}\ (\mathsf{FLAT}\ l) = \mathsf{ALL\_EL}\ \mathsf{NULL}\ l$
$\vdash\ \ \mathsf{SOME\_EL}\ (\mathsf{REVERSE}\ l) = \mathsf{SOME\_EL}\ l$

Figure 4: Example theorems proved by `LIST_CONV`

3.4 The conversion `LIST_CONV`

The proof procedure outlined above is implemented as a conversion named `LIST_CONV`. Its input is a term of the form

CONST1...(CONST2...)

where CONST2 should be one of the following constants: [ ], CONS, SNOC, APPEND, FLAT or REVERSE. It returns an equality whose left-hand side is the input term and whose right-hand side is obtained by specialising an appropriate fold theorem. For example, by calling `LIST_CONV` with the term `LENGTH(APPEND l1 l2)`, Theorem 7 can be derived.

The general FOLDR theorems which `LIST_CONV` currently uses are given in Figure 3. It also uses similar theorems about FOLDL. Figure 4 gives examples that illustrate the range of theorems about system constants that can be derived using `LIST_CONV`. Similar theorems could be derived about user-defined constants. As with all conversions, the theorems are not in closed (universally quantified) form. This allows conversions to be easily composed.

`LIST_CONV` relies on information stored in a database to perform the derivation. Initially, it holds information about the system constants only. It can be augmented to include information about user-defined constants so that more theorems can be derived.

CONST1	CONST2					
	[ ]	CONS	SNOC	APPEND	FLAT	REVERSE
CONS	$\surd$	$\surd$	$\surd$	$\leftrightarrow$	$\times$	$\leftrightarrow$
SNOC	$\surd$	$\surd$	$\surd$	$\leftrightarrow$	$\times$	$\leftrightarrow$
APPEND	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
FLAT	$\surd$	$\surd$	$\surd$	$\surd$	$\leftrightarrow$	$\leftrightarrow$
REVERSE	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\times$
MAP	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\times$
FILTER	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\times$
LENGTH	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\times$
NULL	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\times$
ALL_EL	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
SOME_EL	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
IS_EL	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
SUM	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
AND_EL	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
OR_EL	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
PREFIX	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\times$
SUFFIX	$\surd$	$\times$	$\surd$	$\times$	$\times$	$\times$
FLAT o REVERSE	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\times$

Table 1: Pairs of constants about which `LIST_CONV` can prove theorems

The database contains theorems defining the constants in terms of FOLDR or FOLDL. These theorems should have the form:

$\vdash$ CONST1 $\ldots\ l \ldots$ = FOLD{R/L} $f\ e\ l$

where l is a list variable. The theorems in Figure 2 all have this form.

As shown in previous sub-sections, auxiliary information concerning the step functions and base elements may be required. This information is also stored in the database. In particular, monoid, commutativity, associativity, left identity, right identity, and binary function commutativity theorems are stored. The database can be inspected using the function `list_thm_database` and updated using the function `set_list_thm_database`.

Table 1 shows the theorems that can be proven with no additional information supplied by the user. The symbol $\surd$ indicates that the theorem can be proved, $\times$ that it cannot, and $\leftrightarrow$ that a symmetric version of the theorem can be proved.

The form of the resulting theorem depends on CONST1 and CONST2. Table 2 shows this together with the auxiliary theorems needed. Here we assume CONST1 is defined in terms of FOLDR. A similar table follows for constants defined in terms of FOLDL. The theorem $\vdash$ ASSOC f can be used in place of $\vdash$ FCOMM f f. $\vdash$ MONOID f e can be used at any point instead of $\vdash$ ASSOC f or $\vdash$ LEFT_ID f. This means that rather than needing to store the same information in several different forms, only the most general theorem need be stored.

CONST2	Result	Side conditions
[]	e	NONE
$[x]$	$f\ e\ x$	NONE
CONS $x\ l$	$f\ x$ (CONST1 l)	NONE
SNOC $x\ l$	CONST1 $(f\ e\ x)\ l$	e is a list variable
APPEND $l_1\ l_2$	CONST1 (CONST1 l_1) l_2	e is a list variable
APPEND $l_1\ l_2$	g (CONST1 l_1) (CONST1 l_2)	FCOMM $g\ f$, LEFT_ID $g\ e$
FLAT l	CONST3 (MAP CONST1 l)	FCOMM $g\ f$, LEFT_ID $g\ e$,
		CONST3 l = FOLDR $g\ e\ l$
REVERSE l	CONST1 l	ASSOC f, COMM f
REVERSE l	CONST1 l	$f\ \equiv\ \lambda x\ l.\ h\ (g\ x)\ l$
		ASSOC h, COMM h

Table 2: The behaviour of `LIST_CONV`

`LIST_CONV` first checks if it knows a theorem defining CONST1 in terms of either FOLDR or FOLDL. If it has both, it tries FOLDR first. It then looks for auxiliary information about the step function and base element of CONST1. If appropriate information is found for one of the general fold theorems about CONST2, modus ponens is used to give a rewrite rule. The original term is then rewritten with it and the fold definition. Beta conversion is applied and the term is rewritten using a reversed (possibly different) fold definition from the database.

The advantage of this approach is that it can be used to derive theorems about both system and user defined constants. It is also relatively fast, since the derivation involves only rewriting, beta reduction and possibly modus ponens. The common parts of the proof including the inductive part are effectively precomputed and stored in the theorem about the higher-order fold function. This is a big advantage for a fully-expansive theorem prover such as HOL. The additional overhead in searching for appropriate theorems is very small in comparison to carrying out the inferences.

The disadvantage is that the user must provide theorems about the step function f and the base element e which are sometimes difficult to identify. However, these theorems are usually simpler to prove than the required list theorem and can often be proved automatically using other tools. Also, once added to the database, one piece of auxiliary information can be used to prove several theorems.

`PURE_LIST_CONV` is a version of `LIST_CONV` which does not use the database. All additional information is passed as explicit arguments. `X_LIST_CONV` uses the database but also takes user-supplied information as an argument.

4 Evaluation Conversions

Accompanying the new list theory is a set of evaluation conversions for the list operators. They take as argument a term which applies a particular list operator to a concrete list. They return a theorem equating the original term to the result of evaluating it. For example, the call

```
APPEND_CONV "APPEND [1;2] [3;4]"
```

returns the theorem:

$$\vdash \quad \mathsf{APPEND}[1;2][3;4] = [1;2;3;4] \tag{8}$$

We also provide a general purpose evaluation conversion `list_FOLD_CONV`. It can be used to evaluate both system constants and user-defined constants. As with `LIST_CONV` it is based on FOLDR and FOLDL, requiring a theorem that defines the constant in terms of one of them. This theorem should be passed to `list_FOLD_CONV` as the first argument. It must also be given, as the second argument, an evaluation conversion for the step function of the fold definition.

It first unfolds the input term with the defining theorem. This results in a term in which FOLDR or FOLDL, rather than the user-defined constant, is applied to the concrete list. It then evaluates the term using the conversion `FOLDR_CONV` or `FOLDL_CONV`.

For example, `APPEND_CONV` may be implemented using this general evaluation conversion. Theorem 4 can be passed to `list_FOLD_CONV` as the defining theorem and `ALL_CONV` can be passed to evaluate the step function, CONS. `ALL_CONV` is sufficient here because CONS is a basic constructor and as such does not require further evaluation. Suppose we need to evaluate the APPEND term in the example above, the conversion first unfolds this input to obtain the term FOLDR CONS [3;4] [1;2]. Then, it applies the conversion `FOLDR_CONV ALL_CONV` to this new term. Theorem 8 follows from this together with the equality

$$\vdash \quad \mathsf{APPEND}[1;2][3;4] = \mathsf{FOLDR\ CONS\ [3;4]\ [1;2]}$$

by transitivity.

Thus efficient evaluation conversions can quickly be built for constants defined in terms of the fold operators. The problem is reduced to that of writing an evaluation conversion for the simpler function that the constant is defined in terms of.

5 Conclusions

The new list theory is more comprehensive and coherent than the previous theory. It is in the core system. Incompatible changes to the previous core theory have been kept to a minimum. The new theory uses the higher-order fold functions extensively. This has allowed us to write general purpose proof procedures which can prove theorems about both system *and* user-defined constants. Users of the theory are encouraged to define new constants in terms of FOLDR or FOLDL rather than recursively to make most use of this.

`LIST_CONV` is more than just a lookup table for theorems, since the theorems do not need to have been proven in advance. Indeed, it can prove theorems about constants defined by the user. For system defined constants, most theorems `LIST_CONV` can prove have been stored in the list theory. This was done for consistency. However, as the number of constants increases, the number of theorems that must be stored may become prohibitive. Tools such as `LIST_CONV` which can quickly generate them on the fly offer a viable alternative. This approach also overcomes the problem of remembering the names of theorems, since they are “accessed” by their text rather than by name. This could be used in conjunction with memoisation [2] so that theorems are not repeatedly proved in a session.

LIST_CONV could be extended to prove a wider range of theorems by determining other theorems about the fold functions. Some preliminary work in this direction has been undertaken. By combining LIST_CONV with depth rewriting conversions, more complex theorems than those described here can also be proven. Similar proof procedures could be written based on general theorems about SEG. This would allow theorems about constants such as FIRSTN to be quickly proved. Similar proof procedures could also be written for other data types. This would involve defining constants similar to the fold functions for the new data types which encapsulate the recursive definitions and then determining general theorems about those constants.

Acknowledgements

We acknowledge the contributions of the many people who created the original HOL list theories. We are also grateful to those who responded to the list theory questionnaire. Brian Graham originally pointed out the advantages of basing the theory upon fold left and right, and translated the evaluation conversions to HOL90.

References

[1] R.J. Bird and P. Wadler. *Introduction to functional programming.* Prentice-Hall International, 1988.

[2] Donald Michie. Memo functions and machine learning. *Nature*, 218:19–22, 6th April 1968.

Appendices

A Naming of Theorems

We have attempted to use a coherent naming strategy for constants, their definitions and theorems. Recursive definitions are named after the constant. Non-recursive definitions are given the name of the constant suffixed by _DEF. Names reflect the main constants in the theorem in the left to right order that they appear. A theorem having the following form would have the name CONST1_CONST2.

$\vdash$ CONST1 ... (CONST2 ...) ... = ...

For example, the following theorem is named LENGTH_REVERSE:

$\vdash$ LENGTH(REVERSE l) = LENGTH l

There are also theorems relating the derived constants to the basic constants. Their names are the concatenation of the two constants with an underline(_) in between. For example, the following theorem has the name APPEND_FOLDR.

$\vdash$ APPEND l_1 l_2 = FOLDR CONS l_2 l_1

We have left names and definitions from the old core theory unchanged wherever possible. The only changes are:

- the constant EVERY has been renamed ALL_EL;
- the name of its definition becomes ALL_EL; and
- the names of theorems about EVERY become ALL_EL....

The naming convention we have adopted were already used widely.

B Constant definitions

This appendix lists all constant definitions in the `list` theory.

B.1 Constructors and general functions

$\vdash (\forall l.\, \mathsf{APPEND}\,[\,]\,l = l) \wedge (\forall l_1\, l_2\, h.\, \mathsf{APPEND}\,(\mathsf{CONS}\,h\,l_1)\,l_2 = \mathsf{CONS}\,h\,(\mathsf{APPEND}\,l_1\,l_2))$

$\vdash \forall h\, t.\, \mathsf{CONS}\,h\,t = \mathsf{ABS\_list}$
$\quad ((\lambda m.\, ((m = 0) \Rightarrow h \mid \mathsf{FST}\,(\mathsf{REP\_list}\,t)\,(\mathsf{PRE}\,m))), \mathsf{SUC}\,(\mathsf{SND}\,(\mathsf{REP\_list}\,t)))$

$\vdash (\mathsf{FLAT}\,[\,] = [\,]) \wedge (\forall h\, t.\, \mathsf{FLAT}\,(\mathsf{CONS}\,h\,t) = \mathsf{APPEND}\,h\,(\mathsf{FLAT}\,t))$

$\vdash (\mathsf{LENGTH}\,[\,] = 0) \wedge (\forall h\, t.\, \mathsf{LENGTH}\,(\mathsf{CONS}\,h\,t) = \mathsf{SUC}\,(\mathsf{LENGTH}\,t))$

$\vdash [\,] = \mathsf{ABS\_list}\,((\lambda n.\, \varepsilon e.\, \mathsf{T}), 0)$

$\vdash (\mathsf{REVERSE}\,[\,] = [\,]) \wedge (\forall x\, l.\, \mathsf{REVERSE}\,(\mathsf{CONS}\,x\,l) = \mathsf{SNOC}\,x\,(\mathsf{REVERSE}\,l))$

$\vdash (\forall x.\, \mathsf{SNOC}\,x\,[\,] = [x]) \wedge (\forall x\, x'\, l.\, \mathsf{SNOC}\,x\,(\mathsf{CONS}\,x'\,l) = \mathsf{CONS}\,x'\,(\mathsf{SNOC}\,x\,l))$

B.2 Higher-order functions

$\vdash (\forall P.\, \mathsf{FILTER}\,P\,[\,] = [\,]) \wedge$
$\quad (\forall P\, x\, l.\, \mathsf{FILTER}\,P\,(\mathsf{CONS}\,x\,l) = (P\,x \Rightarrow \mathsf{CONS}\,x\,(\mathsf{FILTER}\,P\,l) \mid \mathsf{FILTER}\,P\,l))$

$\vdash (\forall f\, e.\, \mathsf{FOLDL}\,f\,e\,[\,] = e) \wedge (\forall f\, e\, x\, l.\, \mathsf{FOLDL}\,f\,e\,(\mathsf{CONS}\,x\,l) = \mathsf{FOLDL}\,f\,(f\,e\,x)\,l)$

$\vdash (\forall f\, e.\, \mathsf{FOLDR}\,f\,e\,[\,] = e) \wedge (\forall f\, e\, x\, l.\, \mathsf{FOLDR}\,f\,e\,(\mathsf{CONS}\,x\,l) = f\,x\,(\mathsf{FOLDR}\,f\,e\,l))$

$\vdash (\forall f.\, \mathsf{MAP}\,f\,[\,] = [\,]) \wedge (\forall f\, h\, t.\, \mathsf{MAP}\,f\,(\mathsf{CONS}\,h\,t) = \mathsf{CONS}\,(f\,h)\,(\mathsf{MAP}\,f\,t))$

$\vdash (\forall f.\, \mathsf{MAP2}\,f\,[\,]\,[\,] = [\,]) \wedge$
$\quad (\forall f\, h_1\, t_1\, h_2\, t_2.\, \mathsf{MAP2}\,f\,(\mathsf{CONS}\,h_1\,t_1)\,(\mathsf{CONS}\,h_2\,t_2) = \mathsf{CONS}\,(f\,h_1\,h_2)\,(\mathsf{MAP2}\,f\,t_1\,t_2))$

$\vdash (\forall f\, e.\, \mathsf{SCANL}\,f\,e\,[\,] = [e]) \wedge (\forall f\, e\, x\, l.\, \mathsf{SCANL}\,f\,e\,(\mathsf{CONS}\,x\,l) = \mathsf{CONS}\,e\,(\mathsf{SCANL}\,f\,(f\,e\,x)\,l))$

$\vdash (\forall f\, e.\, \mathsf{SCANR}\,f\,e\,[\,] = [e]) \wedge$
$\quad (\forall f\, e\, x\, l.\, \mathsf{SCANR}\,f\,e\,(\mathsf{CONS}\,x\,l) = \mathsf{CONS}\,(f\,x\,(\mathsf{HD}\,(\mathsf{SCANR}\,f\,e\,l)))\,(\mathsf{SCANR}\,f\,e\,l))$

B.3 Predicates

$\vdash (\forall P.\, \mathsf{ALL\_EL}\,P\,[\,] = \mathsf{T}) \wedge (\forall P\, x\, l.\, \mathsf{ALL\_EL}\,P\,(\mathsf{CONS}\,x\,l) = P\,x \wedge \mathsf{ALL\_EL}\,P\,l)$

$\vdash \mathsf{AND\_EL} = \mathsf{ALL\_EL}\,\mathsf{I}$

$\vdash \forall x\, l.\, \mathsf{IS\_EL}\,x\,l = \mathsf{SOME\_EL}\,(\${=}\,x)\,l$

$\vdash (\mathsf{NULL}\,[\,] = \mathsf{T}) \wedge (\forall h\, t.\, \mathsf{NULL}\,(\mathsf{CONS}\,h\,t) = \mathsf{F})$

$\vdash \mathsf{OR\_EL} = \mathsf{SOME\_EL}\,\mathsf{I}$

$\vdash (\forall P.\, \mathsf{SOME\_EL}\,P\,[\,] = \mathsf{F}) \wedge (\forall P\, x\, l.\, \mathsf{SOME\_EL}\,P\,(\mathsf{CONS}\,x\,l) = P\,x \vee \mathsf{SOME\_EL}\,P\,l)$

$\vdash (\forall l.\, \mathsf{IS\_PREFIX}\, l\, [\,] = \mathsf{T}) \wedge$
$\quad (\forall x\, l.\, \mathsf{IS\_PREFIX}\, [\,]\, (\mathsf{CONS}\, x\, l) = \mathsf{F}) \wedge$
$\quad (\forall x_1\, l_1\, x_2\, l_2.\, \mathsf{IS\_PREFIX}\, (\mathsf{CONS}\, x_1\, l_1)\, (\mathsf{CONS}\, x_2\, l_2) = (x_1 = x_2) \wedge \mathsf{IS\_PREFIX}\, l_1\, l_2)$

$\vdash (\forall l.\, \mathsf{IS\_SUBLIST}\, l\, [\,] = \mathsf{T}) \wedge$
$\quad (\forall x\, l.\, \mathsf{IS\_SUBLIST}\, [\,]\, (\mathsf{CONS}\, x\, l) = \mathsf{F}) \wedge$
$\quad (\forall x_1\, l_1\, x_2\, l_2.\, \mathsf{IS\_SUBLIST}\, (\mathsf{CONS}\, x_1\, l_1)\, (\mathsf{CONS}\, x_2\, l_2) = (x_1 = x_2) \wedge$
$\qquad \mathsf{IS\_PREFIX}\, l_1\, l_2 \vee \mathsf{IS\_SUBLIST}\, l_1\, (\mathsf{CONS}\, x_2\, l_2))$

$\vdash (\forall l.\, \mathsf{IS\_SUFFIX}\, l\, [\,] = \mathsf{T}) \wedge$
$\quad (\forall x\, l.\, \mathsf{IS\_SUFFIX}\, [\,]\, (\mathsf{SNOC}\, x\, l) = \mathsf{F}) \wedge$
$\quad (\forall x_1\, l_1\, x_2\, l_2.\, \mathsf{IS\_SUFFIX}\, (\mathsf{SNOC}\, x_1\, l_1)\, (\mathsf{SNOC}\, x_2\, l_2) = (x_1 = x_2) \wedge \mathsf{IS\_SUFFIX}\, l_1\, l_2)$

$\vdash \forall P\, l.\, \mathsf{PREFIX}\, P\, l = \mathsf{FST}\, (\mathsf{SPLITP}\, (\neg \circ P)\, l)$

$\vdash (\forall P.\, \mathsf{SPLITP}\, P\, [\,] = [\,], [\,]) \wedge$
$\quad (\forall P\, x\, l.\, \mathsf{SPLITP}\, P\, (\mathsf{CONS}\, x\, l) =$
$\qquad (P\, x \Rightarrow [\,], \mathsf{CONS}\, x\, l \mid \mathsf{CONS}\, x\, (\mathsf{FST}\, (\mathsf{SPLITP}\, P\, l)), \mathsf{SND}\, (\mathsf{SPLITP}\, P\, l)))$

$\vdash \forall P\, l.\, \mathsf{SUFFIX}\, P\, l = \mathsf{FOLDL}\, (\lambda l'\, x. (P\, x \Rightarrow \mathsf{SNOC}\, x\, l' \mid [\,]))\, [\,]\, l$

B.4 Segments

$\vdash (\forall l.\, \mathsf{BUTFIRSTN}\, 0\, l = l) \wedge (\forall n\, x\, l.\, \mathsf{BUTFIRSTN}\, (\mathsf{SUC}\, n)\, (\mathsf{CONS}\, x\, l) = \mathsf{BUTFIRSTN}\, n\, l)$

$\vdash \forall l.\, \mathsf{BUTLAST}\, l = \mathsf{SEG}\, (\mathsf{PRE}\, (\mathsf{LENGTH}\, l))\, 0\, l$

$\vdash (\forall l.\, \mathsf{BUTLASTN}\, 0\, l = l) \wedge (\forall n\, x\, l.\, \mathsf{BUTLASTN}\, (\mathsf{SUC}\, n)\, (\mathsf{SNOC}\, x\, l) = \mathsf{BUTLASTN}\, n\, l)$

$\vdash (\forall l.\, \mathsf{EL}\, 0\, l = \mathsf{HD}\, l) \wedge (\forall l\, n.\, \mathsf{EL}\, (\mathsf{SUC}\, n)\, l = \mathsf{EL}\, n\, (\mathsf{TL}\, l))$

$\vdash (\forall l.\, \mathsf{ELL}\, 0\, l = \mathsf{LAST}\, l) \wedge (\forall n\, l.\, \mathsf{ELL}\, (\mathsf{SUC}\, n)\, l = \mathsf{ELL}\, n\, (\mathsf{BUTLAST}\, l))$

$\vdash (\forall l.\, \mathsf{FIRSTN}\, 0\, l = [\,]) \wedge (\forall n\, x\, l.\, \mathsf{FIRSTN}\, (\mathsf{SUC}\, n)\, (\mathsf{CONS}\, x\, l) = \mathsf{CONS}\, x\, (\mathsf{FIRSTN}\, n\, l))$

$\vdash \forall h\, t.\, \mathsf{HD}\, (\mathsf{CONS}\, h\, t) = h$

$\vdash \forall l.\, \mathsf{LAST}\, l = \mathsf{HD}\, (\mathsf{SEG}\, 1\, (\mathsf{PRE}\, (\mathsf{LENGTH}\, l))\, l)$

$\vdash (\forall l.\, \mathsf{LASTN}\, 0\, l = [\,]) \wedge (\forall n\, x\, l.\, \mathsf{LASTN}\, (\mathsf{SUC}\, n)\, (\mathsf{SNOC}\, x\, l) = \mathsf{SNOC}\, x\, (\mathsf{LASTN}\, n\, l))$

$\vdash (\forall k\, l.\, \mathsf{SEG}\, 0\, k\, l = [\,]) \wedge$
$\quad (\forall m\, x\, l.\, \mathsf{SEG}\, (\mathsf{SUC}\, m)\, 0\, (\mathsf{CONS}\, x\, l) = \mathsf{CONS}\, x\, (\mathsf{SEG}\, m\, 0\, l)) \wedge$
$\quad (\forall m\, k\, x\, l.\, \mathsf{SEG}\, (\mathsf{SUC}\, m)\, (\mathsf{SUC}\, k)\, (\mathsf{CONS}\, x\, l) = \mathsf{SEG}\, (\mathsf{SUC}\, m)\, k\, l)$

$\vdash \forall h\, t.\, \mathsf{TL}\, (\mathsf{CONS}\, h\, t) = t$

B.5 List generators

$\vdash (\forall f.\, \mathsf{GENLIST}\, f\, 0 = [\,]) \wedge (\forall f\, n.\, \mathsf{GENLIST}\, f\, (\mathsf{SUC}\, n) = \mathsf{SNOC}\, (f\, n)\, (\mathsf{GENLIST}\, f\, n))$

$\vdash (\forall x.\, \mathsf{REPLICATE}\, 0\, x = [\,]) \wedge (\forall n\, x.\, \mathsf{REPLICATE}\, (\mathsf{SUC}\, n)\, x = \mathsf{CONS}\, x\, (\mathsf{REPLICATE}\, n\, x))$

B.6 Special lists

$\vdash (\mathsf{SUM}\,[\,] = 0) \wedge (\forall h\,t.\,\mathsf{SUM}\,(\mathsf{CONS}\,h\,t) = h + \mathsf{SUM}\,t)$

$$\begin{array}{l}\vdash (\mathsf{UNZIP}\,[\,] = [\,], [\,]) \wedge \\ \quad (\forall x\,l.\,\mathsf{UNZIP}\,(\mathsf{CONS}\,x\,l) = \\ \quad\quad \mathsf{CONS}\,(\mathsf{FST}\,x)\,(\mathsf{FST}\,(\mathsf{UNZIP}\,l)), \mathsf{CONS}\,(\mathsf{SND}\,x)\,(\mathsf{SND}\,(\mathsf{UNZIP}\,l)))\end{array}$$

$\vdash \forall l.\,\mathsf{UNZIP\_FST}\,l = \mathsf{FST}\,(\mathsf{UNZIP}\,l)$

$\vdash \forall l.\,\mathsf{UNZIP\_SND}\,l = \mathsf{SND}\,(\mathsf{UNZIP}\,l)$

$$\begin{array}{l}\vdash (\mathsf{ZIP}\,([\,], [\,]) = [\,]) \wedge \\ \quad (\forall x_1\,l_1\,x_2\,l_2.\,\mathsf{ZIP}\,(\mathsf{CONS}\,x_1\,l_1, \mathsf{CONS}\,x_2\,l_2) = \mathsf{CONS}\,(x_1, x_2)\,(\mathsf{ZIP}\,(l_1, l_2)))\end{array}$$