Applications of Polytypism in Theorem Proving

Konrad Slind[1] and Joe Hurd[2]

[1] School of Computing, University of Utah
[2] Computer Laboratory, University of Cambridge

Abstract. Polytypic functions have mainly been studied in the context of functional programming languages. In that setting, applications of polytypism include elegant treatments of polymorphic equality, prettyprinting, and the encoding and decoding of high-level datatypes to and from low-level binary formats. In this paper, we discuss how polytypism supports some aspects of theorem proving: automated termination proofs of recursive functions, incorporation of the results of metalanguage evaluation, and equivalence-preserving translation to a low-level format suitable for propositional methods. The approach is based on an interpretation of higher order logic types as terms, and easily deals with mutual and nested recursive types.

1 Introduction

When a new datatype is declared in a functional language, or a higher order logic proof system, many functions on that type can be automatically defined. Prime examples of this are maps and folds, of course, but there are also many others: if we think of a datatype declaration facility as a way of introducing a particular shape of tree, common tree operations become immediately definable once such a shape is introduced. Examples include computing the number of nodes in a tree, substitution, hashing, marshalling, *etc.* Such a function is said to be *polytypic*, since its algorithm is the same, modulo the shape of the tree.

We discuss three applications of polytypism to theorem proving:

1. Size functions, which support automated termination proofs.
2. Functions for transporting values from meta-language to object-language.
3. Encoding and decoding functions, which support automated translation from high-level HOL formulas to equivalent boolean formulas.

Our approach is based on an interpretation $[\![\_]\!]_{\Theta,\Gamma}$ of higher order logic types into terms. The interpretation is parameterized by two maps: Θ, which maps type variables; and Γ, which maps type operators.

$$\begin{aligned} [\![v]\!]_{\Theta,\Gamma} &= \Theta(v), && \text{if } v \text{ is a type variable} \\ [\![\,(\tau_1,\ldots,\tau_n)c\,]\!]_{\Theta,\Gamma} &= \Gamma(c)\ \ [\![\tau_1]\!]_{\Theta,\Gamma}\ \cdots\ [\![\tau_n]\!]_{\Theta,\Gamma}, && \text{otherwise} \end{aligned}$$

This interpretation is similar to the semantics of HOL types given by Pitts [15]. Although the interpretation is not itself expressible in the HOL logic, it is expressible in the meta-language (which for us is ML), and uses definitions made in the object logic each time a datatype is defined.

2 Notation and Background Definitions

We use the HOL logic [15] to develop our ideas. The syntax of HOL is based on signatures for types and terms. The type signature Ω assigns arities to type operators. The set of HOL types is the least set closed under the following rules:

type variable. There is a countable set of type variables. Greek letters, *e.g.*, α, β, *etc.* will be used to stand for type variables.
compound type. If c in Ω has arity n, and each of $\tau_1, \ldots \tau_n$ is a type, then $(\tau_1, \ldots, \tau_n)c$ is a type.

A type constant is represented by a 0-ary compound type. The initial types found in Ω: truth values (bool), function space (written $\alpha \to \beta$), and ind, an infinite set of individuals, can be used to definitionally construct a large collection of types. Readers interested in the details of the definition principle used to extend Ω may can consult [15].
Datatypes. A inductive datatype τ declared as an instance of the scheme

$$(\alpha_1, \ldots, \alpha_m)\tau \equiv \mathsf{C}_1\ ty_{11}\ \ldots\ ty_{1k_1}\ | \ldots\ |\ \mathsf{C}_n\ ty_{n1}\ \ldots\ ty_{nk_n},$$

where all the type variables in $ty_{11} \ldots ty_{nk_n}$ are in $\{\alpha_1, \ldots, \alpha_m\}$, denotes the set of all values that can be finitely built up by application of the constructors $\mathsf{C}_1, \ldots, \mathsf{C}_n$. Constructors are injective, and applications of different constructors always yield different values. The type is recursive if any ty_{ij} in the type declaration is $(\alpha_1, \ldots, \alpha_m)\tau$. A characterizing theorem of the following form can be derived for inductive types [21]:

$$\begin{array}{l}
\forall f_1 \ldots f_n.\ \exists! \mathcal{H} : (\alpha_1, \ldots, \alpha_m)\tau \to \beta. \\
\quad \begin{array}{lcl}
\forall x_{11} \ldots x_{1k_1}.\ \mathcal{H}(\mathsf{C}_1\ x_{11} \ldots x_{1k_1}) & = & f_1(\mathcal{H}\ x_{11}) \ldots (\mathcal{H}\ x_{1k_1})\ x_{11} \ldots x_{1k_1}\ \wedge \\
 & \vdots & \qquad\qquad\qquad\qquad \wedge \\
\forall x_{n1} \ldots x_{nk_n}.\ \mathcal{H}(\mathsf{C}_n\ x_{n1} \ldots x_{nk_n}) & = & f_n(\mathcal{H}\ x_{n1}) \ldots (\mathcal{H}\ x_{nk_n})\ x_{n1} \ldots x_{nk_n}.
\end{array}
\end{array}$$

This theorem embodies the primitive recursion principle for functions over the specified type. More complex datatypes, featuring mutual and nested recursion, allow similar theorems to be proved for them [17,18,5]. We will make use of the following types (given in SML syntax):

```
1.   datatype 'a option = NONE | SOME of 'a
2.   datatype 'a list = [] | :: of 'a * 'a list
3.   datatype 'a tree = Node of 'a * 'a tree list
4.   datatype ('a,'b)exp = Var of 'a
                         | Cond  of ('a,'b)bexp * ('a,'b)exp * ('a,'b)exp
                         | App of 'b * ('a,'b)exp list
     and     ('a,'b)bexp = Less of ('a,'b)exp * ('a,'b)exp
                         | And of ('a,'b)bexp * ('a,'b)bexp
                         | Not of ('a,'b)bexp
```

The first two declaration are for the well-known types of options and polymorphic lists, the third is a type of finitely branching trees, and the fourth is a mutually recursive type of expressions and boolean expressions.

Miscellaneous definitions. The K combinator is defined $\mathsf{K} \equiv \boldsymbol{\lambda} xy.x$. The first and second components of a pair are selected by $\mathsf{fst}(x, y) \equiv x$ and $\mathsf{snd}(x, y) \equiv y$ respectively; and the $(\mathsf{SOME}\ x) \equiv x$ returns the element from an option. The function $\mathcal{I}$ applies an interpretation to a term:

$$\mathcal{I}_{\Theta,\Gamma}(M : \tau) = [\![\tau]\!]_{\Theta,\Gamma}(M) \ .$$

Note. Although constructors in ML take a single argument tuple, by default constructors in HOL are curried.

3 Wellfounded Relations for Datatypes

In a logic of total functions such as HOL, the termination of recursive function definitions must be proved. A simple and yet effective approach is to show that the arguments of all recursive calls decrease in size. Therefore, it is common practice to define *size measures* on datatypes. The following formal definitions and theorems provide justification (where $\mathsf{WF} : (\alpha \to \alpha \to \mathsf{bool}) \to \mathsf{bool}$ is a predicate that singles out the *wellfounded* relations):

$$\begin{array}{ll} \mathsf{inv\_image}\ R\ f \equiv \boldsymbol{\lambda} x\ y.\ R\ (f\ x)\ (f\ y) & \vdash \mathsf{WF}(R) \supset \mathsf{WF}\ (\mathsf{inv\_image}\ R\ f) \\ \mathsf{measure} \equiv \mathsf{inv\_image}\ (<) & \vdash \forall f : \alpha \to \mathsf{num}.\ \mathsf{WF}\ (\mathsf{measure}\ f) \end{array}$$

Thus termination of a recursive function over a type τ may be proved by showing that the recursive calls are in the relation $\mathsf{measure}(size_\tau)$. Thinking of an element of a datatype as a tree, it is standard to define its size as one plus the sum of the sizes of the subtrees (leaves are assigned a size of zero).[1] We now encounter a problem: how to define size for *polymorphic* datatypes? For example, it would be easy, natural, and unsatisfactory—for our notion of size, which is the number of internal nodes in a tree—to define the size of a list as its length: length does not capture the size of a list of numbers, or a list of lists. Moreover, how are we to capture, in a single definition, the sizes of lists of elements of types not yet defined? It seems that the notion of size for a polymorphic type would have to know about all types that have been defined in the past, and also all types that could be defined in the future!

We solve this polymorphic quandry by mapping each type variable in the type to a term variable representing a size function. Thus, we intend to define the size of elements of datatype $(\alpha_1, \ldots, \alpha_n)\tau$ as a higher order function parameterized by n size functions, one for each type variable:

$$\tau\mathsf{\_size}\ (f_1 : \alpha_1 \to \mathsf{num}) \ldots (f_n : \alpha_n \to \mathsf{num})\ (x : (\alpha_1, \ldots, \alpha_n)\tau) \equiv \ldots$$

A general way to construct such definitions uses $[\![\_]\!]_{\Theta,\Gamma}$. The idea is to traverse the type, and build a term by replacing type operators by size functions (by use of Γ), and type variables by parameters (by use of Θ). Thus Γ maps previously defined type operators to their associated size functions, and Θ maps $\alpha_1, \ldots, \alpha_n$ to $f_1, \ldots, f_n$.

[1] This seems to be motivated by the desire to have the size of Peano numerals be the identity function.

Definition 1 (Datatype size).

Suppose datatype $(\alpha_1, \ldots, \alpha_n)\tau$ *has been defined, with constructors* $\mathsf{C}_1, \ldots, \mathsf{C}_k$ *in size context* Γ. *Create function variables* $(f_1 : \alpha_1 \to \mathsf{num}), \ldots, (f_n : \alpha_n \to \mathsf{num})$, *and let* Θ *be* $\{\alpha_1 \mapsto f_1, \ldots, \alpha_n \mapsto f_n\}$. *Extend* Γ *with a binding for* $\mathsf{size\_}\tau$*:*

$$\Delta = \boldsymbol{\lambda} tyop.\ \textit{if } tyop = \tau \textit{ then } size\_\tau \textit{ else } \Gamma(tyop).$$

Then define

$$\begin{array}{ll} \mathsf{size\_}\tau\ f_1 \ldots f_n\ \mathsf{C}_i & \equiv 0, \quad \textit{if } \mathsf{C}_i \textit{ is nullary; otherwise,} \\ \mathsf{size\_}\tau\ f_1 \ldots f_n\ (\mathsf{C}_i\ x_1 \ldots x_m) & \equiv 1 + \sum_{i=1}^{m} \mathcal{I}_{\Theta,\Delta}(x_i) \end{array}$$

□

Example 1. Size definitions for our example types:

– Lists:

$$\begin{array}{ll} \mathsf{list\_size}\ f\ [\,] & = 0 \\ \mathsf{list\_size}\ f\ (h :: t) & = 1 + f\ h + \mathsf{list\_size}\ f\ t. \end{array}$$

– Trees:

$$\mathsf{tree\_size}\ f\ (\mathsf{Node}\ x\ \mathit{tlist}) = 1 + fx + \mathsf{list\_size}\ (\mathsf{tree\_size}\ f)\ \mathit{tlist}.$$

– Expressions, boolean expressions, and expression lists:

$$\begin{array}{ll} \mathsf{esize}\ f\ g\ (\mathsf{Var}\ a) & = 1 + f\ a \\ \mathsf{esize}\ f\ g\ (\mathsf{Cond}\ b\ e_1\ e_2) & = 1 + \mathsf{bsize}\ f\ g\ b + \mathsf{esize}\ f\ g\ e_1 + \mathsf{esize}\ f\ g\ e_2 \\ \mathsf{esize}\ f\ g\ (\mathsf{App}\ \mathit{fn}\ \ell) & = 1 + g\ \mathit{fn} + \mathsf{elsize}\ f\ g\ \ell \\ \hline \mathsf{bsize}\ f\ g\ (\mathsf{Less}\ e_1\ e_2) & = 1 + \mathsf{esize}\ f\ g\ e_1 + \mathsf{esize}\ f\ g\ e_2 \\ \mathsf{bsize}\ f\ g\ (\mathsf{And}\ e_1\ e_2) & = 1 + \mathsf{bsize}\ f\ g\ e_1 + \mathsf{bsize}\ f\ g\ e_2 \\ \mathsf{bsize}\ f\ g\ (\mathsf{Not}\ b) & = 1 + \mathsf{bsize}\ f\ g\ b \\ \hline \mathsf{elsize}\ f\ g\ [\,] & = 0 \\ \mathsf{elsize}\ f\ g\ (h :: t) & = 1 + \mathsf{esize}\ f\ g\ h + \mathsf{elsize}\ f\ g\ t \end{array}$$

This approach to defining the size of datatype elements becomes particularly useful when dealing with functions defined over instances of polymorphic datatypes.

Example 2. Consider a polymorphic function $\mathsf{flat} : \alpha\ \mathsf{list}\ \mathsf{list} \to \alpha\ \mathsf{list}$ for removing a level of bracketing from a list:

$$\begin{array}{r} \mathsf{flat}\ [\,] \equiv [\,] \\ \mathsf{flat}\ ([\,] :: \ell) \equiv \mathsf{flat}\ \ell \\ \mathsf{flat}\ ((h :: t) :: \ell) \equiv h :: \mathsf{flat}\ (t :: \ell). \end{array}$$

Note that simply measuring the length of the argument will not prove termination. To show that flat terminates, we first ensure that Γ contains $\mathsf{list} \mapsto \mathsf{list\_size}$, and that $\Theta \equiv \{\alpha \mapsto \mathsf{K}\ 0\}$. Then

$$[\![\alpha\ \mathsf{list}\ \mathsf{list}]\!]_{\Theta,\Gamma} = \mathsf{list\_size}\ (\mathsf{list\_size}\ (\mathsf{K}\ 0)),$$

and proving termination of flat can be done by showing that the recursive calls of flat lie in the relation $\mathsf{measure}\ (\mathsf{list\_size}\ (\mathsf{list\_size}\ (\mathsf{K}\ 0)))$, *i.e.*, making the informal abbreviation $\mathcal{M} \equiv \mathsf{list\_size}\ (\mathsf{K}\ 0)$, by showing

$$\begin{array}{lll}
1.\ \mathsf{list\_size}\ \mathcal{M}\ \ell & < \mathsf{list\_size}\ \mathcal{M}\ ([\,] :: \ell) \\
 & = 1 + \mathcal{M}\ [\,] + \mathsf{list\_size}\ \mathcal{M}\ \ell
\end{array}$$

$$\begin{array}{ll}
2.\ \mathsf{list\_size}\ \mathcal{M}\ (t :: \ell) & = 1 + \mathcal{M}\ t + \mathsf{list\_size}\ \mathcal{M}\ \ell \\
 & < \mathsf{list\_size}\ \mathcal{M}\ ((h :: t) :: \ell) \\
 & = 1 + \mathcal{M}\ (h :: t) + \mathsf{list\_size}\ \mathcal{M}\ \ell \\
 & = 1 + (1 + (\mathsf{K}\ 0)\ h + \mathcal{M}\ t) + \mathsf{list\_size}\ \mathcal{M}\ \ell \\
 & = 1 + (1 + \mathcal{M}\ t) + \mathsf{list\_size}\ \mathcal{M}\ \ell
\end{array}$$

This kind of derivation is straightforward to automate: $[\![-]\!]$ is used to construct a measure on the sizes of recursive calls and then the resulting problem is reduced via rewriting to a problem in linear arithmetic. This has been implemented for several years in the HOL-4 system. Such termination proofs fail either because the termination relation is wrong or because the automated termination condition prover is too weak. For example, termination of the following 'higher order recursion' fails to be proved automatically: the size measure is correct but the termination prover is currently too weak, since it doesn't attempt induction.

Example 3. Consider a function for accumulating the node elements of a tree into a set (using 'foldl', a fold on lists):

$$\mathsf{Nodeset}\ (\mathsf{Node}\ v\ \ell) = \mathsf{foldl}\ (\boldsymbol{\lambda} acc\ t.\ acc \cup \mathsf{Nodeset}\ t)\ \{v\}\ \ell$$

The size measure for this definition is $\mathsf{measure}\ (\mathsf{tree\_size}\ (\mathsf{K}\ 0))$, and the synthesized termination requirement [25, pages 131-133] is

$$\exists R.\ \mathsf{WF}\ R \land \forall v\ \ell\ t.\ \mathsf{mem}\ t\ \ell \supset R\ t\ (\mathsf{Node}\ v\ \ell).$$

Using the size measure as a witness results in a goal

$$\forall \ell\ t.\ \mathsf{mem}\ t\ \ell \supset \mathsf{tree\_size}(\mathsf{K}\ 0)\ t < \mathsf{list\_size}\ (\mathsf{tree\_size}\ (\mathsf{K}\ 0))\ \ell + 1$$

which is provable by induction on ℓ. □

A typical Γ would include at least the following:

type	size definition
$\tau_1 * \tau_2$	$\mathbf{prod\_size}\ f\ g\ (x, y) \equiv f\ x + g\ y$
$\tau_1 + \tau_2$	$\mathbf{sum\_size}\ f\ g\ (\mathsf{INL}\ x) \equiv f\ x$
	$\mathbf{sum\_size}\ f\ g\ (\mathsf{INR}\ y) \equiv g\ y$
bool	$\mathbf{bool\_size}\ x \equiv 0$
num	$\mathbf{num\_size}\ x \equiv x$
option	$\mathbf{option\_size}\ f\ \mathsf{NONE} \equiv 0$
	$\mathbf{option\_size}\ f\ (\mathsf{SOME}\ x) \equiv 1 + (f\ x)$

Remarks. The size of a pair is just the sum of the sizes of the two projections: since pairs are not recursive, it is not useful (for the purposes of termination proofs) to add one to the sum. The size of an element of a sum type is just the size of the injected item: since sum constructors are used as discriminatory tags, any nesting of INL and INR should be ignored in the computation of an object's size. We have found this to be a useful approach when proving termination of mutually recursive functions, which are modelled using sum types [6]. As a map, Γ is partial: if a type constructor is not in Γ, then all elements of that type are deemed to be of a fixed size. For example, functions have a fixed size (zero).

Summary. In this section we have examined a way to automatically generate size measures for datatypes, and sketched a method for automated termination proofs of recursive functions. The approach is naive, especially compared to the work in [11,12]. However, our size measures automatically prove termination for a relatively large class of functions and can be used as the base relation in more powerful relations (multiset, rpo, *etc*), so our work is of general utility.

4 Lifting of Metalanguage Values

Although evaluation by means of deductive steps can be implemented in HOL in an asymptotically efficient way [2], it is still very attractive to be able to execute ground terms—when possible—with a more efficient engine, such as the ML implementation underlying HOL. A recent example of this is [4], in which ML is used to evaluate recursive functions and (some) inductive relations defined in Isabelle/HOL. Other systems that exploit the speed of evaluation in the implementation language are ACL2 and PVS [22,24]. These two systems are based in LISP, the reflective capabilities of which make for a smooth passage back and forth between meta- and object- languages.

However, current logic implementations with an ML-evaluation feature have the drawback that answers computed at the ML level are not automatically lifted back into object-language terms. This is somewhat unsatisfying since we'd like to make use of the following diagram:

$$\begin{tikzcd}
\textbf{HOL} & \mathcal{M} \arrow[r, "\text{Deductive evaluation}"] \arrow[d, "\text{drop}"'] & \vdash \mathcal{M} = \mathcal{N} \\
\textbf{ML} & \mathcal{M} \arrow[r, "\text{ML evaluation}"'] & \mathcal{N} \arrow[u, "\text{lift}"']
\end{tikzcd}$$

Such a system—mapping HOL terms to ML expressions, performing ML evaluation, and lifting the result back to HOL—is useful with large formalizations. For example, in a recent model of the ARM [9,10], deductive evaluation of ARM assembly programs achieves a speed of a few tens of instructions per second.

In contrast, much higher execution speeds for hardware models have been reported with ACL2, which compiles and evaluates formal definitions directly in the underlying LISP implementation. We would like to attain similar speeds in HOL.

It is relatively easy to drop HOL terms—it is just a matter of prettyprinting to ML syntax—but lifting is more difficult, because the lifter depends on the type of the data to be lifted and so can't be written once and forall as an ML function. Of course, if the results of ML execution aren't needed in the object logic, the lifting step can be dispensed with. However, it may be that results computed in ML can find other application, *e.g.*, existential witnesses or counterexamples may be found by external tools, and then lifted to terms in proofs.

Our approach is to interpret a HOL type[2] $\ulcorner\tau\urcorner$ into an ML program which will lift an ML value $M : \tau$ to a HOL term $\ulcorner M : \tau\urcorner$. We will have to modify our interpretation so that it explicitly passes the type. This allows, for example, the lifting of $[] :$ **num list**.

$$\begin{array}{lr}[\![\tau]\!]_{\Theta,\Gamma} = \Theta(\tau), & \text{if } \tau \text{ is a type variable} \\ [\![\tau]\!]_{\Theta,\Gamma} = \Gamma(c)\ \tau\ [\![\tau_1]\!]_{\Theta,\Gamma}\ \cdots\ [\![\tau_n]\!]_{\Theta,\Gamma} & \text{where } \tau = (\tau_1, ..., \tau_n)c\end{array}$$

This is analogous to the difference between primitive recursion and iteration, an issue explored in a slightly different context by Weirich [26].

Definition 2 (Datatype lifting). *Suppose datatype* $(\alpha_1, \ldots, \alpha_n)\tau$ *has been defined in HOL, with constructors* $\mathsf{C}_1, \ldots, \mathsf{C}_k$ *in lifter context* Γ. *Also assume that the same datatype has been declared in ML. Create ML function variables* $(f_1 : \alpha_1 \to \mathsf{term}), \ldots, (f_n : \alpha_n \to \mathsf{term})$, *and let* Θ *be* $\{\alpha_1 \mapsto f_1, \ldots, \alpha_n \mapsto f_n\}$. *Extend* Γ *with a binding for* $\mathsf{lift\_}\tau$*:*

$$\Delta = \boldsymbol{\lambda} tyop.\ \mathit{if}\ tyop = \tau\ \mathit{then}\ \mathit{lift\_}\tau\ \mathit{else}\ \Gamma(tyop).$$

Then define the ML function

$$\mathsf{lift\_}\tau\ \tau\ f_1 \ldots f_n\ (\mathsf{C}_i\ (x_1, \ldots, x_m)) \equiv \ulcorner \mathsf{C}_i\ (\mathcal{I}_{\Theta,\Delta}(x_1)) \ldots (\mathcal{I}_{\Theta,\Delta}(x_m))\ :\ \tau\urcorner.$$

□

Example 4. The lifter $\mathsf{lift\_list} : \mathsf{hol\_type} \to (\alpha \to \mathsf{term}) \to \alpha\ \mathsf{list} \to \mathsf{term}$ for the datatype of lists is :

$$\begin{array}{c}\mathsf{lift\_list}\ ty\ f\ [] \equiv \ulcorner [] : ty\urcorner \\ \mathsf{lift\_list}\ ty\ f\ (h :: t) \equiv \ulcorner f\ h :: \mathsf{lift\_list}\ ty\ f\ t\urcorner.\end{array}$$

The actual definition made in our implementation is more contorted, mainly in order to avoid type instantiations at each node in the list. □

An inference rule ML_EVAL that reduces ground HOL terms by ML evaluation can be easily implemented by dropping the given term $M : \tau$, evaluating M in

[2] The notation $\ulcorner - \urcorner$ is used to distinguish HOL types and terms from ML types and expressions.

ML, and then using τ to synthesize and apply a lifter to the result, yielding a HOL term N. The result is then asserted as an *oracle* theorem $\vdash M = N$ having an attached tag that attests to the ML excursion [23]. Thus theorems generated by ML_EVAL are *weakened* by the meta-language excursion, but the speed-up may be worth it in some cases.

Example 5. The improvement of execution speed for ground formulas is as expected: deductively evaluating (with EVAL) the standard factorial function over the first twenty-one numbers takes 3.6 seconds and 123,520 primitive inference steps.

```
- Count.apply EVAL (Term ‘Map Fact (iota 0 20)‘);
runtime: 3.625s,    gctime: 0.331s,     systime: 3.625s.
HOL primitive inference steps: 123520.
> val it = |- Map Fact (iota 0 20) =
    [1; 1; 2; 6; 24; 120; 720; 5040; 40320; 362880; 3628800; 39916800;
     479001600; 6227020800; 87178291200; 1307674368000; 20922789888000;
     355687428096000; 6402373705728000; 121645100408832000;
     2432902008176640000] : thm
```

In contrast, the expression sent to ML by ML_EVAL

$$\begin{array}{l}\mathsf{lift\_list}\ \ulcorner\mathsf{:\ num\ list}\urcorner\ (\mathsf{lift\_num}\ \ulcorner\mathsf{:\ num}\urcorner)\\ \quad(\mathsf{Map\ Fact\ (iota\ (numML.fromString"0")(numML.fromString"20")))}\end{array}$$

wraps the lifter generated from the type num list around the ML expression generated from the HOL term Map Fact (iota 0 20) and takes 0.03 seconds and one inference step. □

5 Encoding and Decoding

It is common in computer science to create operations that package up high-level data as flat strings of bits, and corresponding operations to unpack strings of bits and recover the high-level data. When this is done to send data over a communication network, it is called marshalling/unmarshalling, but we will use the general terminology *encoding/decoding* or just *coding.* An advantage of encoding high-level data as strings of bits is that operations such as encryption or compression can be uniformly applied to any kind of data. An interesting application of coding in HOL is the mapping of high-level formulas into equivalent quantified boolean formulas suitable for input to the powerful SAT implementations that have recently become popular.

5.1 Encoders

Intuitively, an encoding function can be thought of simply as an injective function $\tau \to$ bool list mapping elements of type τ to lists of booleans. The injectivity condition prevents two elements of τ being encoded as the same list of booleans, and so guarantees that if a list can be decoded then the decoding will be unique.

Encoding functions can be automatically defined when a new datatype is declared, in exactly the same way as the size functions of Section 3.

Definition 3 (Datatype encoding).
Suppose datatype $(\alpha_1, \ldots, \alpha_n)\tau$ *has been defined, with* k *constructors* $\mathsf{C}_1, \ldots, \mathsf{C}_k$ *in encoding context* Γ*. Create function variables* $(f_1 : \alpha_1 \to \mathsf{bool\ list}), \ldots, (f_n : \alpha_n \to \mathsf{bool\ list})$*, and let* Θ *be* $\{\alpha_1 \mapsto f_1, \ldots, \alpha_n \mapsto f_n\}$*. Extend* Γ *with a binding for* $\mathsf{encode\_}\tau$*:*

$$\Delta = \boldsymbol{\lambda} tyop.\ \textit{if } tyop = \tau \textit{ then } \mathsf{encode\_}\tau \textit{ else } \Gamma(tyop).$$

Then define

$$\begin{aligned} &\mathsf{encode\_}\tau\ f_1 \ldots f_n\ (\mathsf{C}_i\ (x_1 : \tau_1) \ldots (x_m : \tau_m)) \\ &\quad \equiv\ \mathsf{marker}\ k\ i\ @\ (\mathcal{I}_{\Theta,\Delta}(x_1))\ @\ \cdots\ @\ (\mathcal{I}_{\Theta,\Delta}(x_m)) \end{aligned}$$

where @ represent the list append function, and $\mathsf{marker}\ k\ i$ *is the* i*th boolean list of length* N *(*N *is the smallest natural number satisfying* $k \leq 2^N$*.)* □

Example 6. The encoding function for the datatype of lists:

$$\begin{aligned} \mathsf{encode\_list}\ f\ [\,] &\equiv [\bot]\ \wedge \\ \mathsf{encode\_list}\ f\ (h :: t) &\equiv \top :: f\ h\ @\ \mathsf{encode\_list}\ f\ t \end{aligned}$$

Lists have two constructors, which are distinguished in each case of encode_list by the prepending of $\mathsf{marker}\ 2\ 0 = [\bot]$ and $\mathsf{marker}\ 2\ 1 = [\top]$. □

The marker lists are designed to be just long enough to be able to distinguish between the datatype constructors. Lists have two constructors, and so the marker lists have length one. A datatype with eight constructors would need marker lists of length three. As the next example shows, nothing needs to be altered in the special case of a datatype with a single constructor.

Example 7. The encoding function for the datatype of trees:

$$\mathsf{encode\_tree}\ e\ (\mathsf{Node}\ a\ b) \equiv e\ a\ @\ \mathsf{encode\_list}\ (\mathsf{encode\_tree}\ e)\ b$$

Since the tree datatype has only one constructor, the encoding function prepends $\mathsf{marker}\ 1\ 0 = [\,]$ to the result (and this gets simplified away). □

The final example we present shows the definition of a custom encoder. Although encoders are automatically defined for every datatype declared, the user may wish to override the automatic definition with an alternative version, or to provide an encoder for a non-datatype.

Example 8. The encoding function for the type num of natural numbers:

$$\begin{aligned} \mathsf{encode\_num}\ n \equiv\ & \text{if } n = 0 \text{ then } [\top;\ \top] \\ & \text{else if } \mathsf{even}\ n \text{ then } \bot :: \mathsf{encode\_num}\ ((n-2)\ \mathsf{div}\ 2) \\ & \text{else } \top :: \bot :: \mathsf{encode\_num}\ ((n-1)\ \mathsf{div}\ 2) \end{aligned}$$

Note that in the even case, recursing with $(n-2)\ \mathsf{div}\ 2$ instead of $n\ \mathsf{div}\ 2$ leads to a more compact encoding. □

A typical environment Γ for encoding functions would include at least:

type	encoder
$\tau_1 * \tau_2$	$\mathbf{encode\_prod}\ f\ g\ (x, y) \equiv f\ x\ @\ g\ y$
$\tau_1 + \tau_2$	$\mathbf{encode\_sum}\ f\ g\ (\mathsf{INL}\ x) \equiv \bot :: f\ x$
	$\mathbf{encode\_sum}\ f\ g\ (\mathsf{INR}\ y) \equiv \top :: g\ y$
bool	$\mathbf{encode\_bool}\ x \equiv [x]$
option	$\mathbf{encode\_option}\ f\ \mathsf{NONE} \equiv [\bot]$
	$\mathbf{encode\_option}\ f\ (\mathsf{SOME}\ x) \equiv \top :: f\ x$
num	**encode_num** (defined above)
τ list	**encode_list** (defined above)

As can be seen, function spaces are omitted completely from this list; we cannot simply return a default value (as we did for size functions) because we require that all encoders are injective functions. On the other hand, we include the list type because lists play such a fundamental role in encoding.

5.2 Decoders: Existence

A decoder for type τ is an algorithm that takes as input a list of booleans and returns an element of type τ. The strategy we will present makes it possible to build decoders in a type-directed way. The key is to think of a decoder for type τ as a function

$$\mathsf{decode\_}\tau : \mathsf{bool\ list} \to (\tau \times \mathsf{bool\ list})\ \mathsf{option}$$

This function tries to 'parse' an input list of booleans into an element of type τ, and if it succeeds then it returns the element of τ *together with the list of booleans that were left over*. If it fails to parse the input list, it signals this by returning NONE. A standard decoding function of type $\mathsf{bool\ list} \to \tau$ can be recovered from $\mathsf{decode\_}\tau$ using the function $\langle\cdot\rangle$, defined as $\langle \mathsf{decode\_}\tau \rangle \equiv \mathsf{fst} \circ \mathsf{the} \circ \mathsf{decode\_}\tau$.

Given some datatype $(\alpha_1, \ldots, \alpha_n)\tau$, we would like to specify $\mathsf{decode\_}\tau$ as the inverse of $\mathsf{encode\_}\tau$. For a given domain predicate P, the $\mathsf{coder}\ P\ e\ d$ property requires that the encoder e and decoder d are mutually inverse:

$$\mathsf{coder}\ P\ e\ d\ \equiv\ \forall\, l, x, t.\ P\ x \supset ((l = e\ x\, @\, t) \iff (d\ l = \mathsf{SOME}\ (x, t)))$$

This allows us to use $\mathsf{encode\_}\tau$ to define the specification of $\mathsf{decode\_}\tau$:

$$\begin{array}{l}\mathsf{coder}\ P_1\ e_1\ d_1 \wedge \cdots \wedge \mathsf{coder}\ P_n\ e_n\ d_n \supset \\ \mathsf{coder}\ (\mathsf{all\_}\tau\ P_1 \ldots P_n)\ (\mathsf{encode\_}\tau\ e_1 \ldots e_n)\ (\mathsf{decode\_}\tau\ d_1 \ldots d_n)\end{array}$$

The function $\mathsf{all\_}\tau$ lifts the predicates $P_i : \alpha_i \to \mathsf{bool}$ to a predicate of the datatype $(\alpha_1, \ldots, \alpha_n)\tau$, and has type

$$\mathsf{all\_}\tau : (\alpha_1 \to \mathsf{bool}) \to \cdots \to (\alpha_n \to \mathsf{bool}) \to (\alpha_1, \ldots, \alpha_n)\tau \to \mathsf{bool}$$

Similarly to $\mathsf{size\_}\tau$ and $\mathsf{encode\_}\tau$ functions, we can use polytypism to define an $\mathsf{all\_}\tau$ function whenever a new datatype is declared.

It turns out that there exists a $\mathsf{decode\_}\tau$ satisfying the above specification precisely when $\mathsf{encode\_}\tau$ is prefix-free on domain $\mathsf{all\_}\tau\ P_1 \ldots P_n$, where

$$\mathsf{prefixfree}\ P\ e\ \equiv\ \forall x, y.\ P\ x \land P\ y \land \mathsf{is\_prefix}\ (e\ x)\ (e\ y) \supset x = y$$

Since every list l satisfies $\mathsf{is\_prefix}\ l\ l$, being prefix-free is a stronger property on $\mathsf{encode\_}\tau$ than being an injective function. Therefore, the following theorem is all we need to show the existence of a $\mathsf{decode\_}\tau$ satisfying the above specification:

$$\begin{array}{l}\mathsf{prefixfree}\ P_1\ e_1 \land \cdots \land \mathsf{prefixfree}\ P_n\ e_n \supset \\ \mathsf{prefixfree}\ (\mathsf{all\_}\tau\ P_1 \ldots P_n)\ (\mathsf{encode\_}\tau\ e_1 \ldots e_n)\end{array}$$

Our scheme of defining encoders using marker lists means that it would not be difficult to provide a polytypic tactic that proves this theorem automatically for every new datatype.

Once we have proved the existence of decoder functions satisfying the above specification, the final step is to pick an arbitrary one (using the axiom of choice) to define a new constant called $\mathsf{decode\_}\tau$.

5.3 Decoders: Recursion Equations

Given a new datatype $(\alpha_1, \ldots, \alpha_n)\tau$, we have shown how to use polytypism to define $\mathsf{decode\_}\tau$ as the inverse of $\mathsf{encode\_}\tau$. This is a useful sanity check, demonstrating that it is always possible to uniquely decode an element of τ that was previously encoded as a boolean list.

Unfortunately, the specification of $\mathsf{decode\_}\tau$ is not in a form that allows us to directly execute it on a boolean list. However, using polytypism once again we can write down a set of recursion equations for $\mathsf{decode\_}\tau$, and then use logical inferences to show that they follow from the specification.

Example 9. The recursion equations for decode_bool:

$$\begin{array}{r}\mathsf{decode\_bool}\ [\,] \equiv \mathsf{NONE}\ \land \\ \mathsf{decode\_bool}\ (h :: t) \equiv \mathsf{SOME}\ (h, t)\end{array}$$

In other words, the only time we fail to decode an element of type bool is when we are given the empty list. □

Deriving the recursion equations for a decoder proceeds by case analysis on the input list, followed by application of the decoder specification and the definition of the corresponding encoder. This proof strategy also works when the recursion equations for a decoder are recursive, as is the case for the list type.

Example 10. The recursion equations for decode_list:

```
reducing $d$ $\supset$
  decode_list $d$ $[\,]$ $\equiv$ NONE $\land$
  decode_list $d$ $(\bot :: l)$ $\equiv$ SOME $([\,], l)$ $\land$
  decode_list $d$ $(\top :: l)$ $\equiv$
    case $d\ l$ of NONE $\to$ NONE
    | SOME $(h, l')$ $\to$ case decode_list $d\ l'$ of NONE $\to$ NONE
                             | SOME $(t, l'')$ $\to$ SOME $(h :: t, l'')$
```

Termination of the recursion equations is ensured by the $\mathsf{reducing}\ d$ side-condition, which requires that the boolean list returned by d is always a sublist of its input (all decoders will satisfy this property). □

From the previous example we can see that in general the recursion equations for $\mathsf{decode\_}\tau$ will be co-recursive and have side-conditions requiring the sub-decoders $d_1, \ldots, d_n$ to satisfy $\mathsf{reducing}$. Such definitions combined with higher-order recursion can produce some entertaining problems, and we finish with a particularly tricky example.

Example 11. The recursion equations for $\mathsf{decode\_tree}$:

$$\begin{array}{l}
\mathsf{reducing}\ d\ \supset \\
\quad \mathsf{decode\_tree}\ d\ l\ \equiv \\
\quad\quad \mathsf{case}\ d\ l\ \mathsf{of}\ \mathsf{NONE}\ \to\ \mathsf{NONE} \\
\quad\quad \mid \mathsf{SOME}\ (a, l')\ \to\ \mathsf{case}\ \mathsf{decode\_list}\ (\mathsf{decode\_tree}\ d)\ l'\ \mathsf{of}\ \mathsf{NONE}\ \to\ \mathsf{NONE} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{SOME}\ (b, l'')\ \to\ \mathsf{SOME}\ (\mathsf{Node}\ a\ b, l'')
\end{array}$$

Because we defined $\mathsf{decode\_tree}$ as the inverse of $\mathsf{encode\_tree}$, we can first prove that $\mathsf{reducing}\ (\mathsf{decode\_tree}\ d)$ holds, and then make use of the $\mathsf{decode\_list}$ recursion equations to derive the $\mathsf{decode\_tree}$ recursion theorems. However, without such a back door it appears to be difficult to define this kind of function in HOL, and we present $\mathsf{decode\_tree}$ as a challenge example for termination proving. □

Example 12. Executing an encoder followed by a decoder.

$$\begin{array}{l}
\mathsf{encode\_list}\ \mathsf{encode\_num}\ [1;\ 2] = [\top;\ \top;\ \bot;\ \top;\ \top;\ \top;\ \bot;\ \top;\ \top;\ \bot] \\
\mathsf{decode\_list}\ \mathsf{decode\_num}\ [\top;\ \top;\ \bot;\ \top;\ \top;\ \top;\ \bot;\ \top;\ \top;\ \bot] = \mathsf{SOME}\ ([1;\ 2], [\,])
\end{array}$$

5.4 Converting Formulas to Boolean Form

Up to this point we have only applied encoders to ground datatype terms, but to create versions of problems suitable for SAT solvers or model checkers we need to convert whole formulas to boolean form, in particular handling variables properly. Our translation methodology has much in common with that used in bounded model checking to map temporal logic formulas to propositional logic [1]. We first define the *boolean propagation theorems* which we use to replace the functions and predicates in the formula with boolean versions.

Definition 4 (Boolean propagation theorems). *For every function*

$$f : \tau_1 \to \cdots \to \tau_n \to \tau$$

of arity n that occurs in formulas to be converted to boolean form, we must define a version

$$\hat{f} : \mathsf{bool\ list} \to \cdots \to \mathsf{bool\ list} \to \mathsf{bool\ list}$$

that operates on boolean lists. The boolean propagation theorem for f is

$$f\ (\langle \mathsf{decode\_}\tau_1 \rangle\ x_1) \ldots (\langle \mathsf{decode\_}\tau_n \rangle\ x_n) = \langle \mathsf{decode\_}\tau \rangle\ (\hat{f}\ x_1 \ldots x_n).$$

Similarly, for each predicate $P : \tau_1 \to \cdots \to \tau_n \to \mathsf{bool}$,
we define a boolean version $\hat{P} : \mathsf{bool\ list} \to \cdots \to \mathsf{bool\ list} \to \mathsf{bool}$
and prove the theorem $P(\langle \mathsf{decode\_}\tau_1 \rangle\ x_1) \ldots (\langle \mathsf{decode\_}\tau_n \rangle\ x_n) = \hat{P}\ x_1 \ldots x_n$.
□

Secondly, we show how to convert quantifiers over high-level datatypes to quantifiers over boolean variables.

Definition 5 (Boolean variable introduction). *We define two predicates: the first selects elements of a type* τ *that encode to a particular length of boolean list, and the second selects boolean lists that are decodable.*

$$\begin{aligned} \mathsf{width}\ d\ n\ x &\equiv \exists\, l.\ (\mathsf{length}\ l = n) \wedge (d\ l = \mathsf{SOME}\ (x, [\,])) \\ \mathsf{decodable}\ d\ l &\equiv \exists\, x.\ d\ l = \mathsf{SOME}\ (x, [\,]) \end{aligned}$$

Now we can prove the following quantifier conversion theorems:

$$\begin{aligned} (\forall\, x.\ \mathsf{width}\ d\ n\ x \supset P\ x) &\equiv \forall\, l.\ (\mathsf{length}\ l = n) \wedge \mathsf{decodable}\ d\ l \supset P\ (\langle d \rangle\ l) \\ (\exists\, x.\ \mathsf{width}\ d\ n\ x \wedge P\ x) &\equiv \exists\, l.\ (\mathsf{length}\ l = n) \wedge \mathsf{decodable}\ d\ l \wedge P\ (\langle d \rangle\ l) \end{aligned}$$

We assume that all quantifiers over type τ *are restricted by a* $\mathsf{width\ decode\_}\tau\ n$ *predicate where* n *is a concrete natural number.* □

By repeatedly applying boolean variable introduction and the boolean propagation theorems throughout a formula over high-level datatypes, it will be converted into a formula over boolean lists. Only the subformulas containing free variables will be left in their original form. The final step is to reduce the quantifiers over boolean lists to quantifiers over boolean variables, and we do this with the following rewrites:

$$\begin{aligned} (\forall\, l.\ \mathsf{length}\ l = 0 \quad\ \ \supset P\ l) &\equiv P\ [\,] \\ (\forall\, l.\ \mathsf{length}\ l = \mathsf{suc}\ n \supset P\ l) &\equiv \forall\, l.\ \mathsf{length}\ l = n \supset \forall\, b.\ P\ (b :: l) \\ (\exists\, l.\ \mathsf{length}\ l = 0 \quad\ \ \wedge P\ l) &\equiv P\ [\,] \\ (\exists\, l.\ \mathsf{length}\ l = \mathsf{suc}\ n \wedge P\ l) &\equiv \exists\, l.\ \mathsf{length}\ l = n \wedge \exists\, b.\ P\ (b :: l) \end{aligned}$$

5.5 Example: Missionaries and Cannibals

We end our discussion of encoding and decoding with a classic example: three missionaries and three cannibals initially stand on the left bank of a river, and there is a boat available that can carry two people. The aim is get the whole party to the right bank of the river, without ever getting in the situation where the cannibals outnumber the missionaries on either bank. Gordon [13] solved a generalized version of this problem using an embedding of a BDD package into HOL-4. Here we show how to convert the transition relation of the standard problem into boolean form. Following Gordon, we will represent the state as a triple (m, c, b) where $m \leq 3$ is the number of missionaries on the left bank, $c \leq 3$ is the number of cannibals on the left bank, and b is a boolean that is true whenever the boat is on the left bank. A transition is possible between states s and s' whenever the following formula holds:[3]

$$\begin{array}{ll}
\multicolumn{2}{l}{\exists\, m.\ m \le 3 \wedge \exists\, c.\ c \le 3 \wedge \exists\, b.\ \exists\, m'.\ m' \le 3 \wedge \exists\, c'.\ c' \le 3 \wedge \exists\, b'.} \\
\quad (s = (m, c, b)) \wedge (s' = (m', c', b')) \wedge & \text{[the states are well-formed]} \\
\quad b' = \neg b \wedge & \text{[the boat switches banks]} \\
\quad (m' = 0 \vee c' \le m') \wedge & \text{[left bank not outnumbered]} \\
\quad (m' = 3 \vee m' \le c') \wedge & \text{[right bank not outnumbered]} \\
\quad \text{if } b \text{ then} & \\
\qquad m' \le m \wedge c' \le c \wedge & \\
\qquad m' + c' + 1 \le m + c \le m' + c' + 2 & \left[\begin{array}{l}\text{if the boat starts on} \\ \text{the left, 1 or 2 people} \\ \text{travel from left to right}\end{array}\right] \\
\quad \text{else} & \\
\qquad m \le m' \wedge c \le c' \wedge & \\
\qquad m + c + 1 \le m' + c' \le m + c + 2 & \left[\begin{array}{l}\text{else if the boat starts on} \\ \text{the right, 1 or 2 people} \\ \text{travel from right to left}\end{array}\right]
\end{array}$$

To convert this formula to boolean form, we fix an encoding bnum of numbers as fixed-length bitstrings, and for concrete numbers n permit $\exists x.\ x \le n \wedge p\ x$ as syntactic sugar for $\exists x.\ \mathsf{width}\ (\mathsf{decode\_bnum}\ k)\ k\ x \wedge p\ x$, where k is the smallest number satisfying $n < 2^k$. Then we apply our conversion algorithm to obtain

$$\begin{array}{l}
\exists\, m_0, m_1.\ [m_0;\, m_1] \mathrel{\hat{\le}} [\top;\, \top] \ \wedge\ \exists\, c_0, c_1.\ [c_0;\, c_1] \mathrel{\hat{\le}} [\top;\, \top] \wedge \\
\exists\, m_0', m_1'.\ [m_0';\, m_1'] \mathrel{\hat{\le}} [\top;\, \top] \ \wedge\ \exists\, c_0', c_1'.\ [c_0';\, c_1'] \mathrel{\hat{\le}} [\top;\, \top] \ \wedge\ \exists\, b'. \\
\quad s = (\langle \mathsf{decode\_bnum} \rangle\ [m_0;\, m_1],\ \langle \mathsf{decode\_bnum} \rangle\ [c_0;\, c_1],\ \neg b') \wedge \\
\quad s' = (\langle \mathsf{decode\_bnum} \rangle\ [m_0';\, m_1'],\ \langle \mathsf{decode\_bnum} \rangle\ [c_0';\, c_1'],\ b') \wedge \\
\quad ([m_0';\, m_1'] \mathrel{\hat{=}} [\,] \ \vee\ [c_0';\, c_1'] \mathrel{\hat{\le}} [m_0';\, m_1']) \wedge \\
\quad ([m_0';\, m_1'] \mathrel{\hat{=}} [\top;\, \top] \ \vee\ [m_0';\, m_1'] \mathrel{\hat{\le}} [c_0';\, c_1']) \wedge \\
\quad \text{if } \neg b' \text{ then} \\
\qquad [m_0';\, m_1'] \mathrel{\hat{\le}} [m_0;\, m_1] \ \wedge\ [c_0';\, c_1'] \mathrel{\hat{\le}} [c_0;\, c_1] \wedge \\
\qquad [m_0';\, m_1'] \mathbin{\hat{+}} [c_0';\, c_1'] \mathrel{\hat{<}} [m_0;\, m_1] \mathbin{\hat{+}} [c_0;\, c_1] \wedge \\
\qquad [m_0;\, m_1] \mathbin{\hat{+}} [c_0;\, c_1] \mathrel{\hat{\le}} [m_0';\, m_1'] \mathbin{\hat{+}} [c_0';\, c_1'] \mathbin{\hat{+}} [\bot;\, \top] \\
\quad \text{else} \\
\qquad [m_0;\, m_1] \mathrel{\hat{\le}} [m_0';\, m_1'] \ \wedge\ [c_0;\, c_1] \mathrel{\hat{\le}} [c_0';\, c_1'] \wedge \\
\qquad [m_0;\, m_1] \mathbin{\hat{+}} [c_0;\, c_1] \mathrel{\hat{<}} [m_0';\, m_1'] \mathbin{\hat{+}} [c_0';\, c_1'] \wedge \\
\qquad [m_0';\, m_1'] \mathbin{\hat{+}} [c_0';\, c_1'] \mathrel{\hat{\le}} [m_0;\, m_1] \mathbin{\hat{+}} [c_0;\, c_1] \mathbin{\hat{+}} [\bot;\, \top]
\end{array}$$

The only non-boolean parts are the subformulas containing the free s, s' variables. The variable b is no longer present, since it was simplified away to $\neg b'$ during the boolean conversion. If we wanted to take this formula a step further, then we could use Gordon's HolBddLib to symbolically execute these boolean versions of the arithmetic operations, and end up with a BDD representing the transition relation as a pure propositional formula.

6 Related Work

Polytypism has been investigated in the functional programming world for about a decade, in various guises. Hinze [20] gives a nice introduction of an approach

[3] We use the convention that primed variables refer to the state after the transition.

similar to ours. One thrust of research into polytypism is to provide user-level interfaces for polytypic functions; recently, also polytypic *datatypes* such as tries have been investigated [19]. Another avenue of research investigates the use of polytypism in compilation of advanced language features [26]. Our approach to size function definitions appeared in [25].

Capretta [7] showed how to encode datatypes internally in Type Theory, after which polytypic functions may be defined over the encoding. Melham's approach to datatypes [21] could also be cast in this mode: when a single 'large' type of trees is used to encode a class of 'small' types, one could hope that general functions over the large type might be customized to versions over the small types.

Our work on lifting ML values to HOL terms is similar in spirit to work on Normalization by Evaluation, initiated by Berger and Schwichtenberg [3]. This surprising work showed that functions may also be lifted (this is called *readback* in the literature). This work has been taken up by researchers in Type Directed Partial Evaluation [8]. In recent work, Grégoire and Leroy modify the OCAML interpreter so that it implements strong reduction for proof checking in Coq [16]. This relies on a readback function that doesn't follow the structure of types.

Work in model checking has stimulated much interest in automated boolean encodings of temporal logic formulas and transition relations [1]. A handcrafted propositional encoding in HOL of formulas over high-level types *e.g.*, finite sets of pairs of numbers, can be found in [14].

7 Conclusions and Future Work

We have shown how an interpretation of HOL types into terms supports some useful proof automation activities in the HOL-4 system. The interpretation is simple, easy to apply, and deals with all types definable by our datatype package. In our approach, a polytypic function is represented as a derived definition principle.

We have not as yet attempted to provide a user-level interface for defining polytypic functions. However, it seems plausible to let the user instantiate $\mathcal{G}$ in the following scheme:

$$Fn\ (\mathsf{C}_i\ (x_1, ..., x_n)) = \mathcal{G}\ \mathsf{C}_i\ [\mathcal{I}_{\Theta,\Delta}(x_1), \ldots, \mathcal{I}_{\Theta,\Delta}(x_n)]$$

Thus $\mathcal{G}$ would be an ML function expecting a constructor term and a list of terms (the result of applying the interpretation to the arguments of the constructor) and returning a term. This could be used to build each clause in a primitive recursive definition over a datatype. Unfortunately, this scheme would not be general enough to automatically define encoders, which need to know how many constructors a type has.

Acknowledgements. Thanks to Mike Gordon for comments on an intermediate draft. Hurd is supported by EPSRC project GR/R27105/01.

References

1. A.Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu, *Bounded model checking*, To appear in Advances in Computers, Number 58.
2. Bruno Barras, *Proving and computing in HOL*, Theorem Proving in Higher Order Logics, 13th International Conference, TPHOLs 2000, Portland, Oregon, USA, August 14-18, 2000, Proceedings (Mark Aagaard and John Harrison, eds.), Lecture Notes in Computer Science, vol. 1869, Springer, 2000, pp. 17–37.
3. U. Berger and H. Schwichtenberg, *An inverse of the evaluation functional for typed λ-calculus*, Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science LICS'91 (Amsterdam), IEEE Computer Society Press, July 1991, pp. 203–211.
4. Stefan Berghofer and Tobias Nipkow, *Executing higher order logic*, Types for Proofs and Programs (TYPES 2000) (P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, eds.), Lecture Notes in Computer Science, vol. 2277, Springer Verlag, 2002, pp. 24–40.
5. Stefan Berghofer and Markus Wenzel, *Inductive datatypes in HOL - lessons learned in Formal-Logic Engineering*, Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'99) (Nice) (Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, eds.), LNCS, no. 1690, Springer-Verlag, 1999.
6. Richard Boulton and Konrad Slind, *Automatic derivation and application of induction schemes for mutually recursive functions*, Proceedings of the First International Conference on Computational Logic (CL2000) (London, UK), July 2000.
7. Venanzio Capretta, *Recursive families of inductive types*, Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000 (J. Harrison and M. Aagaard, eds.), Lecture Notes in Computer Science, vol. 1869, Springer-Verlag, 2000, pp. 73–89.
8. A. Filinski, *Normalization by evaluation for the computational lambda calculus*, Typed Lambda Calculi and Applications 5th International Conference, TLCA 2001 (Krakow, Poland) (S. Abramsky, ed.), LNCS, vol. 2044, Springer Verlag, May 2001.
9. A. Fox, *A HOL specification of the ARM instruction set architecture*, Tech. Report 545, University of Cambridge Computer Laboratory, June 2001.
10. ______, *Formal verification of the ARM6 micro-architecture*, Tech. Report 548, University of Cambridge Computer Laboratory, November 2002.
11. Juergen Giesl, *Termination analysis for functional programs using term orderings*, Proceedings of the 2nd International Static Analysis Symposium (Glasgow, Scotland), Springer-Verlag, 1995.
12. Jürgen Giesl, *Automatisierung von terminieringsbeweisen für rekursiv defininierte algorithmen*, Ph.D. thesis, Technische Hochshule Darmstadt, 1995.
13. Michael J. C. Gordon, *Programming combinations of deduction and BDD-based symbolic calculation*, LMS Journal of Computation and Mathematics **5** (2002), 56–76.
14. Mike Gordon, *PuzzleTool: an example of programming computation and deduction*, Theorem Proving in Higher Order Logics, 15th International Conference, TPHOLs 2002, Hampton, Virginia, USA, August 2002, Proceedings (V. A Carreno, C. A. Munoz, and S. Tahar, eds.), Lecture Notes in Computer Science, vol. 2410, Springer, 2002, pp. 214–229.
15. Mike Gordon and Tom Melham, *Introduction to HOL, a theorem proving environment for higher order logic*, Cambridge University Press, 1993.

16. Benjamin Grégoire and Xavier Leroy, *A compiled implementation of strong reduction*, International Conference on Functional Programming 2002, ACM Press, 2002.
17. E. L. Gunter, *A broader class of trees for recursive type definitions for HOL*, Higher Order Logic Theorem Proving and its Applications: 6th International Workshop (HUG'93) (J. J. Joyce and C.-J. H. Seger, eds.), Lecture Notes in Computer Science, no. 780, Springer-Verlag, Vancouver, B.C., August 11-13 1994, pp. 141–154.
18. John Harrison, *Inductive definitions: automation and application*, Proceedings of the 1995 International Workshop on Higher Order Logic theorem proving and its applications (Aspen Grove, Utah) (E. Thomas Schubert, Phillip J. Windley, and James Alves-Foss, eds.), LNCS, no. 971, Springer-Verlag, 1995, pp. 200–213.
19. R. Hinze, J. Jeuring, and A. Loeh, *Type-indexed data types*, Mathematics of Program Construction 6th International Conference, MPC 2002 Proceedings (Dagstuhl Castle, Germany) (E.A. Boiten and B. Moeller, eds.), LNCS, no. 2386, Springer Verlag, July 2002, pp. 98–114.
20. Ralf Hinze, *A new approach to generic functional programming*, 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00) (Boston, Massachusetts), ACM Press, 2000.
21. Tom Melham, *Automating recursive type definitions in higher order logic*, Current Trends in Hardware Verification and Automated Theorem Proving (Graham Birtwistle and P.A. Subrahmanyam, eds.), Springer-Verlag, 1989, pp. 341–386.
22. J Moore, *Symbolic simulation: An ACL2 approach*, Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design (FM-CAD'98) (G. Gopalakrishnan and P. Windley, eds.), vol. LNCS 1522, Springer-Verlag, November 1998, pp. 334–350.
23. M. Norrish and K. Slind, *A thread of HOL development*, The Computer Journal **45** (2002), no. 1, 37–45.
24. N. Shankar, *Static analysis for safe destructive updates in a functional language*, Logic Based Program Synthesis and Transformation, 11th International Workshop, LOPSTR 2001, Paphos, Cyprus, November 28-30, 2001, Selected Papers (Alberto Pettorossi, ed.), Lecture Notes in Computer Science, vol. 2372, Springer Verlag, 2001, pp. 1–24.
25. Konrad Slind, *Reasoning about terminating functional programs*, Ph.D. thesis, Institut für Informatik, Technische Universität München, 1999, `http://tumb1.biblio.tu-muenchen.de/publ/diss/in/1999/slind.html`.
26. Stephanie Weirich, *Higher-order intensional type analysis*, Programming Languages and Systems: 11th European Symposium on Programming, ESOP 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002 (Daniel Le Métayer, ed.), 2002, pp. 98–114.