Boolification: Encoding High-Level Types as Strings of Bits

Joe Hurd
joe.hurd@cl.cam.ac.uk

University of Cambridge

Joint work with Konrad Slind, University of Utah
Contents

- **Introduction**
- Encoders
- Decoders
- Converting Formulas to Boolean Form
- Conclusion
Introduction

- Encode high-level data as bitstrings, and decode later.

\[ \text{type } \tau \xrightarrow{\text{encode}} \text{bitstrings} \xrightarrow{f} 100101 \cdots \xrightarrow{\text{decode}} \text{bitstrings} \]

- The operation \( f \) could be:
  - transferring data over a network;
  - saving and restoring the state of an interpreter;
  - or compressing, storing, and later decompressing.
Introduction

● **Motivation:** translate HOL goals to boolean form for
  ● SAT solvers (Gordon’s HolSatLib),
  ● BDD reasoning (Gordon’s HolBddLib)
  ● and model checkers (Amjad).

● Need: encoders and decoders for HOL types $\tau$.

● Could do this by hand for each application.

● **Better:** automatic definition of verified encoders and decoders whenever new datatypes are declared.
  ● Will explain how in this talk.
  ● **Warning:** not everything is implemented yet.

● Requires uniform procedures for operating on all HOL types: this is called *polytypism*. 
Encoders

- A $\tau$-encoder is an injective function $\tau \rightarrow \text{bool list}$.
  - The injectivity condition guarantees that decoding is unique whenever it is possible.

- Encoder for natural numbers:

  $$\text{encode\_num } n$$
  $$= \quad \text{if } n = 0 \text{ then } [\top; \top]$$
  $$\quad \text{else if } \text{even } n \text{ then } \bot :: \text{encode\_num } ((n - 2) \div 2)$$
  $$\quad \text{else } \top :: \bot :: \text{encode\_num } ((n - 1) \div 2)$$

- Use extra parameters to handle polymorphic types:

  $$\text{encode\_option } f \text{ NONE } = [\bot]$$
  $$\text{encode\_option } f \text{ (SOME } x) = \top :: f x$$
Polytypism in HOL

- Use an interpretation $\llbracket \cdot \rrbracket^{\Theta, \Gamma}$ of HOL types into terms:

$$
\llbracket \alpha \rrbracket^{\Theta, \Gamma} = \Theta(\alpha) \quad \text{if } \alpha \text{ is a type variable}
$$

$$
\llbracket (\tau_1, \ldots, \tau_n)c \rrbracket^{\Theta, \Gamma} = \Gamma(c) \ \llbracket \tau_1 \rrbracket^{\Theta, \Gamma} \cdots \llbracket \tau_n \rrbracket^{\Theta, \Gamma} \quad o/w
$$

- This scheme cannot be expressed as a higher-order logic function.

- We express it as a meta-language (ML) function.

- Developed by Slind for automatically defining size functions to support well-founded recursion.
Suppose datatype \((\alpha_1, \ldots, \alpha_n) \tau\) (with constructors \(C_1, \ldots, C_k\)) has been declared in encoder context \(\Gamma\).

Define \(\Theta = \{\alpha_1 \mapsto f_1, \ldots, \alpha_n \mapsto f_n\}\).

- The \(f_i : \alpha_i \rightarrow \text{bool} \) list are new function variables.

Extend \(\Gamma\) with a binding for \(\text{encode}_\tau\):

\[
\lambda \text{tyop. if } \text{tyop} = \tau \text{ then } \text{encode}_\tau f_1 \ldots f_n \text{ else } \Gamma(\text{tyop}).
\]

Then define

\[
\text{encode}_\tau f_1 \ldots f_n (C_i \ (x_1 : \tau_1) \ldots (x_m : \tau_m)) = \text{marker} \ k \ i \ @ \ [[\tau_1]]_{\Theta, \Gamma} \ x_1 \ @ \ \cdots \ @ \ [[\tau_m]]_{\Theta, \Gamma} \ x_m
\]

where \(\text{marker} \ k \ i\) is the \(i\)th boolean list of length \(\lceil \log k \rceil\).
Example Encoders

- datatype bool = False | True

  encode_bool False = [⊥] ∧
  encode_bool True = [⊤]

- datatype 'a list = [] | :: of 'a * 'a list

  encode_list f [] = [⊥] ∧
  encode_list f (h :: t) = ⊤ :: f h @ encode_list f t

- datatype tree = Node of tree list

  encode_tree (Node ts) = encode_list encode_tree ts

- All automatically generated. √
Contents

- Introduction
- Encoders
- **Decoders**
- Converting Formulas to Boolean Form
- Conclusion
Decoders

- A $\tau$-decoder ‘parses’ boolean lists into elements of $\tau$:

$$\text{decode}_{\tau} : \text{bool list} \rightarrow (\tau \times \text{bool list}) \text{ option}$$

- Use $\langle \cdot \rangle$ to recover a standard decoding function of type

$$\text{bool list} \rightarrow \tau:\n$$

$$\langle \text{decode}_{\tau} \rangle = \text{fst} \circ \text{the} \circ \text{decode}_{\tau}$$

- The decoder for booleans:

$$\text{decode}_{\text{bool}} [] = \text{NONE} \land$$
$$\text{decode}_{\text{bool}} (h :: t) = \text{SOME} (h, t)$$
Decoders: Existence

- The coder $p e d$ property requires that the encoder $e$ and decoder $d$ are mutually inverse on domain $p$:

$$\forall l, x, t. \ p \ x \ \Rightarrow \ (l = e \ x \ @ \ t \ \iff \ d \ l = \text{SOME} \ (x, t))$$

- Now use $\text{encode}_{\tau}$ to define the specification of $\text{decode}_{\tau}$:

$$\text{coder} \ p_1 \ e_1 \ d_1 \land \cdots \land \text{coder} \ p_n \ e_n \ d_n \ \Rightarrow$$

$$\text{coder} \ (\text{all}_{\tau} \ p_1 \ldots p_n) \ (\text{encode}_{\tau} \ e_1 \ldots e_n) \ (\text{decode}_{\tau} \ d_1 \ldots d_n)$$

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

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

- When is there a $\text{decode}_{\tau}$ satisfying this specification?
Decoders: Existence

- Say an encoder $e$ is prefixfree on $p$ whenever

$$\forall x, y. p x \land p y \land \text{is_prefix}(e x) (e y) \Rightarrow x = y$$

- **Note:** prefixfree is a stronger property than injectivity.

- There exists a $\text{decode}_{\tau}$ satisfying the decoder specification whenever $\text{encode}_{\tau}$ satisfies:

$$\text{prefixfree } p_1 e_1 \land \cdots \land \text{prefixfree } p_n e_n \Rightarrow$$
$$\text{prefixfree } (\text{all}_{\tau} p_1 \ldots p_n) (\text{encode}_{\tau} e_1 \ldots e_n)$$

- **In progress:** prove datatype encoders are prefixfree.

- **Definition step:** use axiom of choice to pick an arbitrary $\text{decode}_{\tau}$ satisfying decoder specification.
Decoders: Recursion Equations

- We define $\text{decode}_T$ as the inverse of $\text{encode}_T$.
  - This provides a useful sanity check on $\text{encode}_T$.
- But we also want recursion equations for $\text{decode}_T$.
  - This will allow us to evaluate $\text{decode}_T$ in the logic.
- We derive the recursion equations of $\text{decode}_T$.
  - The specification of $\text{decode}_T$ has all the information.
- The decoder for products shows the typical shape:

\[
\text{decode}_\text{prod} \ f \ g \ l =
\begin{cases}
  \text{NONE} & \text{if } f \ l \ = \ \text{NONE} \\
  \text{SOME} \ (x, l') & \text{if } f \ l \ = \ \text{SOME} \ (x, l') \\
  \text{SOME} \ ((x, y), l'') & \text{if } f \ l \ = \ \text{SOME} \ ((x, y), l'')
\end{cases}
\]
Decoders: Recursion Equations

- The list decoder is recursive:
  
  \[
  \text{reducing } d \Rightarrow \\
  \text{decode\_list } d \ [\ ] = \text{NONE} \quad \land \\
  \text{decode\_list } d \ (\bot :: l) = \text{SOME} \ ([], l) \quad \land \\
  \text{decode\_list } d \ (\top :: l) = \\
  \quad \text{case } d \ l \text{ of NONE } \rightarrow \text{NONE} \\
  \quad \mid \text{SOME} \ (h, l') \rightarrow \text{case decode\_list } d \ l' \text{ of NONE } \rightarrow \text{NONE} \\
  \quad \mid \text{SOME} \ (t, l'') \rightarrow \text{SOME} \ (h :: t, l'')
  \]

- The sub-decoder \( d \) must satisfy reducing:
  - the bool list returned by \( d \) must be a sublist of its input.
  - This ensures termination of the recursion equations.
Recall: \( \text{datatype \ tree} = \text{Node of \ tree \ list} \)

Here is the decoder for the tree datatype:

\[
\text{decode}_\text{tree} \ l = \\
\begin{cases} 
\text{case \ decode}_\text{list} \ \text{decode}_\text{tree} \ l \ \text{of} \ \text{NONE} & \rightarrow \ \text{NONE} \\
| \ \text{SOME} \ (ts, l') & \rightarrow \ \text{SOME} \ (\text{Node} \ ts, l') 
\end{cases}
\]

To derive these recursion equations:
1. we first prove reducing \( \text{decode}_\text{tree} \);
2. and then use the recursion equations for \( \text{decode}_\text{list} \).

But step 1 relies on \( \text{decode}_\text{tree} \) being already defined.

Put forward \( \text{decode}_\text{tree} \) as a challenge problem for defining functions in an interactive theorem prover.
Decoders: Example

- At this point we have the recursion equations for both encoders and decoders.
- Can evaluate them using logical inference:

\[
\text{encode} \_\text{list} \ \text{encode} \_\text{num} \ [1; 2] = \\
[T; T; \bot; T; T; T; \bot; T; T; \bot]
\]

\[
\text{decode} \_\text{list} \ \text{decode} \_\text{num} \ [T; T; \bot; T; T; T; \bot; T; T; \bot] = \\
\text{SOME} \ ([1; 2], [])
\]
Contents

- Introduction
- Encoders
- Decoders
- Converting Formulas to Boolean Form
- Conclusion
We now present two steps to convert formulas to equivalent quantified boolean formulas (QBF):

1. Replace quantifiers of arbitrary type with quantifiers over boolean variables.
2. Replace functions and predicates with versions operating on boolean lists.
Boolean Variable Introduction

- Define a ‘fixed-width’ predicate:

\[
\text{width } d \ n \ x \iff \exists l. \ \text{length } l = n \land d \ l = \text{SOME } (x, [])
\]

- First convert all quantifiers to be over boolean lists:

\[
(\forall x. \ \text{width } d \ n \ x \Rightarrow p \ x) \iff \forall l. \ (\text{length } l = n) \Rightarrow p \ (\langle d \rangle \ l)
\]

\[
(\exists x. \ \text{width } d \ n \ x \land p \ x) \iff \exists l. \ (\text{length } l = n) \land p \ (\langle d \rangle \ l)
\]

- Then convert all quantifiers to be over booleans:

\[
(\forall l. \ \text{length } l = 0 \Rightarrow p \ l) \iff p \ []
\]

\[
(\forall l. \ \text{length } l = \text{suc } n \Rightarrow p \ l) \iff \forall l. \ \text{length } l = n \Rightarrow \forall b. \ p \ (b :: l)
\]

\[
(\exists l. \ \text{length } l = 0 \land p \ l) \iff p \ []
\]

\[
(\exists l. \ \text{length } l = \text{suc } n \land p \ l) \iff \exists l. \ \text{length } l = n \land \exists b. \ p \ (b :: l)
\]
Boolean Propagation Theorems

- Suppose the following $n$-ary function occurs in formulas:

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

- We must define a version operating on boolean lists:

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

- The **boolean propagation theorem** for $f$ is

  $$ f \left( \langle \text{decode}_{\tau_1} \rangle x_1 \right) \cdots \left( \langle \text{decode}_{\tau_n} \rangle x_n \right) $$

  $$ = \langle \text{decode}_{\tau} \rangle \left( \hat{f} \ x_1 \cdots x_n \right) $$

- Similarly for each $n$-ary predicate.
Missionaries & Cannibals

- Three missionaries and three cannibals on left bank of river.
- Have a boat that can hold up to two people.
- Cannibals must never outnumber missionaries on either bank.
- **Goal:** get everyone to right bank of river.
Missionaries & Cannibals

\[ \exists m. m \leq 3 \land \exists c. c \leq 3 \land \exists b. \exists m'. m' \leq 3 \land \exists c'. c' \leq 3 \land \exists b'. \]

\[ (s = (m, c, b)) \land (s' = (m', c', b')) \land \]

\[ b' = \neg b \land \]

\[ (m' = 0 \lor c' \leq m') \land \]

\[ (m' = 3 \lor m' \leq c') \land \]

if \( b \) then
\[ m' \leq m \land c' \leq c \land \]
\[ m' + c' + 1 \leq m + c \leq m' + c' + 2 \]
else
\[ m \leq m' \land c \leq c' \land \]
\[ m + c + 1 \leq m' + c' \leq m + c + 2 \]

[the states are well-formed]
[the boat switches banks]
[left bank not outnumbered]
[right bank not outnumbered]

if the boat starts on
the left, 1 or 2 people
travel from left to right

else if the boat starts on
the right, 1 or 2 people
travel from right to left
Missionaries & Cannibals

\[ \exists m_0, m_1. [m_0; m_1] \preceq [T; T] \land \exists c_0, c_1. [c_0; c_1] \preceq [T; T] \land \\
\exists m'_0, m'_1. [m'_0; m'_1] \preceq [T; T] \land \exists c'_0, c'_1. [c'_0; c'_1] \preceq [T; T] \land \exists b'. \\
s = (\langle \text{decode bnum} \rangle [m_0; m_1], \langle \text{decode bnum} \rangle [c_0; c_1], \neg b') \land \\
s' = (\langle \text{decode bnum} \rangle [m'_0; m'_1], \langle \text{decode bnum} \rangle [c'_0; c'_1], b') \land \\
([m'_0; m'_1] \equiv [\top] \lor [c'_0; c'_1] \preceq [m'_0; m'_1]) \land \\
([m'_0; m'_1] \equiv [T; T] \lor [m'_0; m'_1] \preceq [c'_0; c'_1]) \land \\
\text{if } \neg b' \text{ then} \\
[m'_0; m'_1] \preceq [m_0; m_1] \land [c'_0; c'_1] \preceq [c_0; c_1] \land \\
[m'_0; m'_1] + [c'_0; c'_1] \preceq [m_0; m_1] + [c_0; c_1] \land \\
[m_0; m_1] + [c_0; c_1] \preceq [m'_0; m'_1] + [c'_0; c'_1] + [\bot; T] \\
\text{else} \\
[m_0; m_1] \preceq [m'_0; m'_1] \land [c_0; c_1] \preceq [c'_0; c'_1] \land \\
[m_0; m_1] + [c_0; c_1] \preceq [m'_0; m'_1] + [c'_0; c'_1] \land \\
[m'_0; m'_1] + [c'_0; c'_1] \preceq [m_0; m_1] + [c_0; c_1] + [\bot; T] \]
Contents

- Introduction
- Encoders
- Decoders
- Converting Formulas to Boolean Form
- Conclusion
Conclusion

- Have shown how to define compositional encoders and decoders in a systematic way.
- Encoders are automatically defined when datatype is declared.
- Automatic definition of decoders present more problems.
  - Showed a possible approach for such a proof tool.
- Converting formulas to boolean form is partly automated.
  - Would be nice if HOL kept track of boolean versions of functions.
- Related work: Hinze’s *generic functional programming*. 