Boolification: Encoding High-Level Types as Strings of Bits

Joe Hurd

joe.hurd@cl.cam.ac.uk

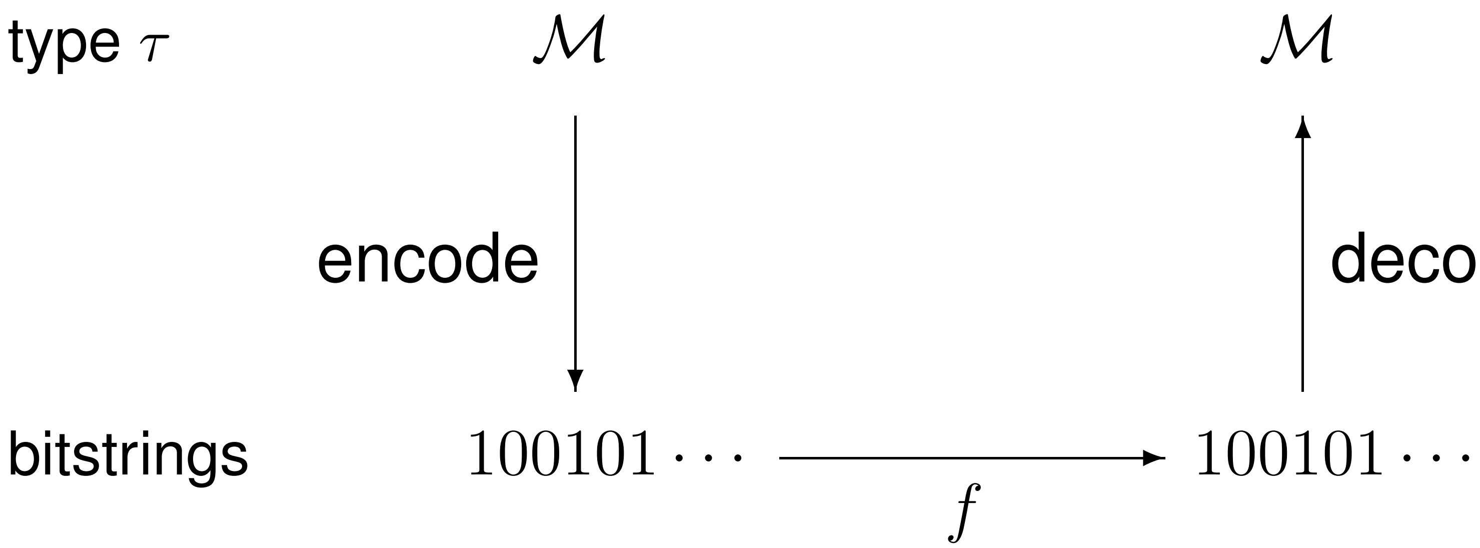
University of Cambridge

Joint work with Konrad Slind, University of Utah

Contents

Introduction

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



- 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 τ.
- 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*.

Contents

Encoders

- A τ-encoder is an injective function $\tau \to$ bool list.
  - The injectivity condition guarantees that decoding is unique whenever it is possible.
- Encoder for natural numbers:

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

- Use extra parameters to handle polymorphic types:

$$
\begin{aligned}
\text{encode\_option } f \text{ NONE} &= [\bot] \\
\text{encode\_option } f \text{ (SOME } x) &= \top :: f\ x
\end{aligned}
$$

Polytypism in HOL

- Use an interpretation $[\![\cdot]\!]_{\Theta,\Gamma}$ of HOL types into terms:

$$
\begin{array}{rcll}
[\![\alpha]\!]_{\Theta,\Gamma} & = & \Theta(\alpha) & \text{if } \alpha \text{ is a type variable} \\
[\![(\tau_1, \ldots, \tau_n)c]\!]_{\Theta,\Gamma} & = & \Gamma(c) \ [\![\tau_1]\!]_{\Theta,\Gamma} \cdots [\![\tau_n]\!]_{\Theta,\Gamma} & o/w
\end{array}
$$

- 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.

Polytypic Encoders

- Suppose datatype $(\alpha_1, \ldots, \alpha_n)\tau$ (with constructors $\mathsf{c}_1, \ldots, \mathsf{c}_k$) has been declared in encoder context Γ.
- Define $\Theta = \{\alpha_1 \mapsto f_1, \ldots, \alpha_n \mapsto f_n\}$.
  - The $f_i : \alpha_i \to \mathsf{bool\ list}$ are new function variables.
- Extend Γ with a binding for $\mathsf{encode\_}\tau$:

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

- Then define

$$\begin{aligned} &\mathsf{encode\_}\tau\ f_1 \ldots f_n\ (\mathsf{c}_i\ (x_1 : \tau_1) \ldots (x_m : \tau_m)) \\ = \ &\mathsf{marker}\ k\ i\ @\ [\![\tau_1]\!]_{\Theta,\Gamma}\ x_1\ @\ \cdots\ @\ [\![\tau_m]\!]_{\Theta,\Gamma}\ x_m \end{aligned}$$

where $\mathsf{marker}\ k\ i$ is the ith boolean list of length $\lceil \log\ k \rceil$.

Example Encoders

- `datatype bool = False | True`

$$\begin{aligned} \text{encode\_bool False} &= [\bot] \quad \wedge \\ \text{encode\_bool True} &= [\top] \end{aligned}$$

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

$$\begin{aligned} \text{encode\_list } f \text{ []} &= [\bot] \quad \wedge \\ \text{encode\_list } f\ (h :: t) &= \top :: f\ h\ @\ \text{encode\_list } f\ t \end{aligned}$$

- `datatype tree = Node of tree list`

$$\text{encode\_tree } (\text{Node } ts) = \text{encode\_list encode\_tree } ts$$

- All automatically generated. $\surd$

Contents

Decoders

- A τ-decoder 'parses' boolean lists into elements of τ:

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

- Use $\langle\cdot\rangle$ to recover a standard decoding function of type bool list $\longrightarrow \tau$:

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

- The decoder for booleans:

$$\begin{aligned} \text{decode\_bool } [\,] &= \text{NONE} \quad \wedge \\ \text{decode\_bool } (h :: t) &= \text{SOME } (h, t) \end{aligned}$$

Decoders: Existence

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

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

- Now use encode_τ to define the specification of decode_τ:

$$\mathsf{coder}\ p_1\ e_1\ d_1 \wedge \cdots \wedge \mathsf{coder}\ p_n\ e_n\ d_n \Rightarrow$$
$$\mathsf{coder}\ (\mathsf{all\_}\tau\ p_1 \ldots p_n)\ (\mathsf{encode\_}\tau\ e_1 \ldots e_n)\ (\mathsf{decode\_}\tau\ d_1 \ldots d_n)$$

  - The function all_τ lifts the predicates $p_i : \alpha_i \to$ bool to a predicate of the datatype $(\alpha_1, \ldots, \alpha_n)\tau$, and has type

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

- When is there a decode_τ 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 decode_τ satisfying the decoder specification whenever encode_τ 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 decode_τ satisfying decoder specification.

Decoders: Recursion Equations

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

$$
\begin{array}{l}
\text{decode\_prod } f\ g\ l\ = \\
\text{case } f\ l \text{ of NONE } \rightarrow \text{ NONE} \\
\mid \text{SOME } (x, l') \rightarrow \text{ case } g\ l' \text{ of NONE } \rightarrow \text{ NONE} \\
\qquad\qquad\qquad\qquad\qquad \mid \text{SOME } (y, l'') \rightarrow \text{ SOME } ((x, y), l'')
\end{array}
$$

Decoders: Recursion Equations

- The list decoder is recursive:

$$\begin{array}{l}
\textsf{reducing}\ d\ \Rightarrow \\
\quad \textsf{decode_list}\ d\ [\,]\ =\ \textsf{NONE}\quad \wedge \\
\quad \textsf{decode_list}\ d\ (\bot :: l)\ =\ \textsf{SOME}\ ([\,], l)\quad \wedge \\
\quad \textsf{decode_list}\ d\ (\top :: l) = \\
\qquad \textsf{case}\ d\ l\ \textsf{of NONE}\ \longrightarrow\ \textsf{NONE} \\
\qquad \mid \textsf{SOME}\ (h, l')\ \longrightarrow\ \textsf{case decode_list}\ d\ l'\ \textsf{of NONE}\ \longrightarrow\ \textsf{NONE} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \textsf{SOME}\ (t, l'')\ \longrightarrow\ \textsf{SOME}\ (h :: t, l'')
\end{array}$$

- 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.

Decoders: Recursion Equations

- Recall: `datatype tree = Node of tree list`
- Here is the decoder for the tree datatype:

$$\begin{aligned} \text{decode\_tree}\ l\ &= \\ &\quad \text{case decode\_list decode\_tree}\ l\ \text{of NONE} \rightarrow \text{NONE} \\ &\quad \mid \text{SOME}\ (ts, l') \rightarrow \text{SOME}\ (\text{Node}\ ts, l') \end{aligned}$$

- To derive these recursion equations:
  1. we first prove reducing decode_tree;
  2. and then use the recursion equations for decode_list.
- But step 1 relies on decode_tree being already defined.
- Put forward decode_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:

$$\text{encode\_list encode\_num } [1;\ 2] = \\ [\top;\ \top;\ \bot;\ \top;\ \top;\ \top;\ \bot;\ \top;\ \top;\ \bot]$$

$$\text{decode\_list decode\_num } [\top;\ \top;\ \bot;\ \top;\ \top;\ \top;\ \bot;\ \top;\ \top;\ \bot] = \\ \text{SOME } ([1;\ 2], [\,])$$

Contents

Converting Formulas to Boolean Form

- 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

$$\begin{aligned} & f\ (\langle \text{decode\_}\tau_1 \rangle\ x_1) \ldots (\langle \text{decode\_}\tau_n \rangle\ x_n) \\ & = \langle \text{decode\_}\tau \rangle\ (\hat{f}\ x_1 \ldots x_n) \end{aligned}$$

- 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

$\exists\, m.\; m \le 3 \wedge \exists\, c.\; c \le 3 \wedge \exists\, b.\; \exists\, m'.\; m' \le 3 \wedge \exists\, c'.\; c' \le 3 \wedge \exists\, b'.$

$(s = (m, c, b)) \wedge (s' = (m', c', b')) \wedge$	[the states are well-formed]
$b' = \neg b \wedge$	[the boat switches banks]
$(m' = 0 \vee c' \le m') \wedge$	[left bank not outnumbered]
$(m' = 3 \vee m' \le c') \wedge$	[right bank not outnumbered]
if b then $m' \le m \wedge c' \le c \wedge$ $m' + c' + 1 \le m + c \le m' + c' + 2$	[if the boat starts on the left, 1 or 2 people travel from left to right]
else $m \le m' \wedge c \le c' \wedge$ $m + c + 1 \le m' + c' \le m + c + 2$	[else if the boat starts on the right, 1 or 2 people travel from right to left]

Missionaries & Cannibals

$$\exists\, m_0, m_1.\ [m_0;\ m_1]\ \hat{\leq}\ [\top;\ \top]\ \wedge\ \exists\, c_0, c_1.\ [c_0;\ c_1]\ \hat{\leq}\ [\top;\ \top]\ \wedge$$

$$\exists\, m'_0, m'_1.\ [m'_0;\ m'_1]\ \hat{\leq}\ [\top;\ \top]\ \wedge\ \exists\, c'_0, c'_1.\ [c'_0;\ c'_1]\ \hat{\leq}\ [\top;\ \top]\ \wedge\ \exists\, b'.$$

$$s = (\langle \textsf{decode\_bnum} \rangle\ [m_0;\ m_1],\ \langle \textsf{decode\_bnum} \rangle\ [c_0;\ c_1],\ \neg b')\ \wedge$$

$$s' = (\langle \textsf{decode\_bnum} \rangle\ [m'_0;\ m'_1],\ \langle \textsf{decode\_bnum} \rangle\ [c'_0;\ c'_1],\ b')\ \wedge$$

$$([m'_0;\ m'_1]\ \hat{=}\ \textsf{[]}\ \vee\ [c'_0;\ c'_1]\ \hat{\leq}\ [m'_0;\ m'_1])\ \wedge$$

$$([m'_0;\ m'_1]\ \hat{=}\ [\top;\ \top]\ \vee\ [m'_0;\ m'_1]\ \hat{\leq}\ [c'_0;\ c'_1])\ \wedge$$

if $\neg b'$ then

$$[m'_0;\ m'_1]\ \hat{\leq}\ [m_0;\ m_1]\ \wedge\ [c'_0;\ c'_1]\ \hat{\leq}\ [c_0;\ c_1]\ \wedge$$

$$[m'_0;\ m'_1]\ \hat{+}\ [c'_0;\ c'_1]\ \hat{<}\ [m_0;\ m_1]\ \hat{+}\ [c_0;\ c_1]\ \wedge$$

$$[m_0;\ m_1]\ \hat{+}\ [c_0;\ c_1]\ \hat{\leq}\ [m'_0;\ m'_1]\ \hat{+}\ [c'_0;\ c'_1]\ \hat{+}\ [\bot;\ \top]$$

else

$$[m_0;\ m_1]\ \hat{\leq}\ [m'_0;\ m'_1]\ \wedge\ [c_0;\ c_1]\ \hat{\leq}\ [c'_0;\ c'_1]\ \wedge$$

$$[m_0;\ m_1]\ \hat{+}\ [c_0;\ c_1]\ \hat{<}\ [m'_0;\ m'_1]\ \hat{+}\ [c'_0;\ c'_1]\ \wedge$$

$$[m'_0;\ m'_1]\ \hat{+}\ [c'_0;\ c'_1]\ \hat{\leq}\ [m_0;\ m_1]\ \hat{+}\ [c_0;\ c_1]\ \hat{+}\ [\bot;\ \top]$$

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*.