Translating Z into HOL

Roger Bishop Jones

ICL Defence Systems

1. INTRODUCTION

This document describes a means of translating a subset of the specification language Z into higher order logic (HOL), and documents special ML procedures developed to facilitate this process. The first issue is merely annotated code rather than a careful exposition, and is issued for reference by people inspecting documents DTC/RBJ/035, DTC/RBJ/044 and DTC/RBJ/045. Definitions and brief explanations of those features of the translation into HOL which are not given in those papers may be found here.

A subsequent issue of this document may provide a more careful explanation of how to use the facilities, which will be separated from the annotated code.

```ml
new_theory '041';;
new_parent 'infra';;
```

2. TYPE CONSTRUCTION

2.1. Simple Type Construction

Where a new type constructor is to be defined which corresponds exactly to an already constructable polytype the basic type definition facility in HOL can be extended to provide automatic proof of existence and automatic definition of a constructor function, together with proof of appropriate properties.

This facility is provided by the function `sim_type_def` defined below. `sim_type_def` takes just two parameters (paired), which specify the name of the new type constructor and the representation type. The type is then introduced using the function `\x:'rep_type.T` to determine the domain of representations, and a function `mk_type_name` is defined of type `(rep_type->type_name)`. Polytypical representation types are handled correctly.

To save a small bit of theorem proving on each use we prove the theorem:

\[ \forall x:*.(T)x \]

Which is then instantiated to the `rep_type` before being used to define the new type constructor.
ML
let exists_thm = TAC_PROOF (\x. EXIST_TAC "\x:(T)x", EVERY [EXIST_TAC "\x:(T)\x", CONV_TAC BETA_CONV]);;

let sim_type_def (tcon_name, rep_type) =

Now we use the HOL type creation mechanism to introduce the new type constructor. This returns a theorem of the form:

|- ONE_ONE REP_tcon \land (\forall f. IS_SUM_REP f = (\exists f'. f = REP_tcon f'))

ML
let is_rep = "\x:rep_type.T"
and rep_thm = INST_TYPE [rep_type, ";;" ] exists_thm
in let rep_thm = new_type_definition (tcon_name, is_rep, rep_thm)

Now we define the abstraction function mk_tcon as the inverse of the representation function REP_tcon (using the epsilon operator).

ML
and mk_def_n = 'mk_' tcon_name 'DEF'
and rep_type = hd (snd (dest_type (type_of is_rep)))
in let mk_v = mk_var ('mk_' tcon_name, rep_type)
and abs_type = mk_type (tcon_name, tyvars is_rep)
in let rep_tcon_t = mk_type ('fun', [abs_type; rep_type])
and mk_tcon_t = mk_type ('fun', [rep_type; abs_type])
in let rep_tcon_c = mk_const ('REP_' tcon_name, rep_tcon_t)
and mk_tcon_v = mk_var ('mk_' tcon_name, mk_tcon_t)
in let mk_tcon_DEF = new_definition (mk_def_n,
"''mk_tcon_v f rep_type = \mu s. f = ''rep_tcon_c s"")
in let mk_tcon_c = mk_const ('mk_' tcon_name, mk_tcon_t)
in (rep_thm, mk_tcon_DEF, rep_tcon_c);

2.2. General Type Constructions
The following procedure provides an enhanced facility for the definition of new type constructors. The code was obtained by cannibalising Tom Melham’s 'sum' type construction code.

ML
let tcon (tcon_name, is_rep, rep_thm) =

First we use the HOL type creation mechanism to introduce the new type constructor. This returns a theorem of the form:

|- ONE_ONE REP_tcon \land (\forall f. IS_SUM_REP f = (\exists f'. f = REP_tcon f'))
ML
let rep_thm = new_type_definition (tcon_name, is_rep, rep_thm)

We then split up the parts of the theorem asserting that the representation function is 1:1 and onto.

ML
in let ONTO_REP_tcon = CONJUNCT2 rep_thm
and ONE_ONE_REP_tcon = REWRITE_RULE [definition 'bool' 'ONE_ONE_DEF'] (CONJUNCT1 rep_thm)

Now we define the abstraction function ABS_tcon as the inverse of the representation function REP_tcon (using the epsilon operator).

ML
and abs_def_n = 'ABS_' ^ tcon_name ^ '_DEF'
and rep_type = hd (snd (dest_type (type_of is_rep)))
in let abs_v = mk_var ('ABS_' ^ tcon_name, rep_type)
and abs_type = mk_type (tcon_name, tyvars is_rep)
in let rep_tcon_t = mk_type ('fun', [abs_type; rep_type])
and abs_tcon_t = mk_type ('fun', [rep_type; abs_type])
in let rep_tcon_c = mk_const ('REP_' ^ tcon_name, rep_tcon_t)
and abs_tcon_v = mk_var ('ABS_' ^ tcon_name, abs_tcon_t)
in let ABS_tcon_DEF = new_definition (abs_def_n, "abs_tcon_v (f:rep_type) = µs. f = rep_tcon_c s")
in let abs_tcon_c = mk_const ('ABS_' ^ tcon_name, abs_tcon_t)

Next we prove that REP_tcon is the left inverse of ABS_tcon (for elements of the representation type that are tcon representations).

ML
in let REP_ABS_THM =
TAC_PROOF([[], "'is_rep (f:rep_type) ∈
(\"rep_tcon_c("abs_tcon_c f) = f")",
REWRITE_TAC [ONTO_REP_tcon; ABS_tcon_DEF] THEN
DISCH_THEN (ACCEPT_TAC o SYM o SELECT_RULE))

We prove that everything given by the representation function REP_tcon lies in that part of the representing type which contains tcon representations.
Next we show that ABS_tcon is the left inverse of REP_tcon. I.e. that:

\[ |- \text{ABS}_tcon(\text{REP}_tcon s) = s \]

We prove that ABS_tcon is one-to-one:

\[ |- (\text{abs}_tcon \ c = \text{abs}_tcon \ c’) = (\text{rep}_tcon \ c = \text{rep}_tcon \ c’) \]

The function returns a list of the theorems proven:

\[ [ \text{rep}_thm; \text{ONT}_tcon; \text{ON}_tcon; \text{AB}_tcon; \text{RE}_thm; \text{IS}_tcon; \text{AB}_thm; \text{ON}_one ]; \]

3. SEQUENCES

We propose to use the type ‘list’ for sequences, primarily because the syntax available for constructing HOL lists matches that for Z sequences.
The following two functions should be used to introduce recursive definitions of functions over lists. They are used in a manner analogous to the similar functions for recursive definitions over number.

For the sake of brevity (in future code):

```ml
let list_rec_def = new_list_rec_definition
and infix_list_rec_def = new_infix_list_rec_definition;;
```

### 3.1. Useful functions over lists

#### 3.1.1. append

```ml
infix_list_rec_def('append',"
($(append:* list → *list → *list) [] lst = lst)
∧ ($append (CONS a l) lst = CONS a ($append l lst))");
```

#### 3.1.2. filter

The following function filters a list using a predicate.

```ml
list_rec_def('filter',"(filter [] (p:* → bool) = []
∧ (∀a:* l:* list)p = (p a) => CONS a (filter l p) | filter l p)");
```

### 4..partial functions

#### 4.1. The type "unit"

We define the type "unit".

```ml
let unit_rep = "λ(x:bool).x=T"
in let unit_app = "(λ(x:bool).x=T)T"
in let vl1 = BETA_CONV unit_app
in let vl2 = SYM vl1
in let vl3 = INST_TYPE ["bool"] EQ_REFL
in let vl4 = SPEC "T:bool" vl3
in let vl5 = EQ_MP vl2 vl4
in let vl6 = EXISTS ("∃(x:bool).(λ(x:bool).x=T)x", "T") vl5
in new_type_definition('unit',unit_rep, vl6);;
```
A possible value is either a unit or some other value, i.e. it has a sum type. A partial function is then a total function yielding possible values.

```ml
let pfun_thms = sim_type_def ('pfun',":*->(unit\+\**)\");;
```

A partial function should be applied using `apf`.

```ml
new_infix_definition('apf',"apf (f:(*,**)pfun) (a:*\") = OUTR (REP_pfun f a)\");;
```

To test whether a partial function is defined for some argument use `pdef`.

```ml
new_infix_definition('pdef',"pdef (f:(*,**)pfun) (a:*\") = ISR (REP_pfun f a)\");;
```

We need to define functional composition of pfuns. For this purpose we introduce four functions:

- Normal (total) function composition.
- Composition of partial functions.
- Composition of a partial function (on the left) with a total function.
- Composition of a partial function (on the right) with a total function.

```ml
new_infix_definition('o','$o (left:*mid->*new) (right:*old->*mid) (arg:*old) = left (right arg)\");;
new_infix_definition('pf_o','$$pf_o (left:(*mid,*new)pfun) (right:*old->*mid) (arg:*old) = left apf (right arg)\");;
% new_infix_definition('o_pf','$o_pf (left:*mid->*new) (right:(*old,*mid),pfun) = mk_pfun \lambda (arg:*old).left (right arg)\");;
%```

To facilitate construction of such functions we define a generic undefined object `undef` and an injection for defined values `pvalue`.

```ml
new_definition('pf_u',"(pf_u:(unit\+*cod)) = INL \mu x:unit.T\");;
new_definition('pf_val',"(pf_val:*cod\to(unit\+*cod)) p = INR p\");;
new_definition('pf_empty',"(pf_empty:(*dom,*cod)pfun) = mk_pfun (\lambda x.pf_u)\");;
```
Two further functions are defined here, one for deleting an object from the domain of a partial function, and the other for adding or replacing an object.

```ml
new_infix_definition('pf_add', "pf_add (f:(*dom,*cod)pfun) (x:*dom) (v:*cod) = mk_pfun (\arg:*dom). (arg=x) => pf_val v |
                f pdef arg => pf_val (f apf arg) | pf_u)");;
new_infix_definition('pf_del', "pf_del (f:(*dom,*cod)pfun) (x:*dom) = mk_pfun (\arg:*dom). (arg=x) => pf_u |
                f pdef arg => pf_val (f apf arg) | pf_u)");;
```

5. HOL LISTING OF THEORY 041