Recursive Data Types in HOL (temp extensions)

Roger Bishop Jones

ICL Defence Systems

1. INTRODUCTION

```
ML
new_theory ‘051‘;;
new_parent ‘042‘;;
```

```
ML
load (‘042‘,false);;
```

```
ML
z_loading := false;;
```

```
ML
z_loading := true;;
```

2. CONSTRUCTING EXISTENTIAL WITNESSES

For each type a witness must be constructed.

To do this the types must be processed in an appropriate order. A type is ready to have a witness constructed under the following conditions:

Disjoint Union
: A disjoint union is ready if any of its components has a witness.

Labelled Product
: A labelled product is ready if all of its components has a witness.

Void and Primitive components
: always have witnesses. The witnesses are constructed during the initialisation phase.

The code in this section does not merely construct witnesses. In the process of constructing the witnesses it constructs a plan for a proof that the witnesses satisfy the predicates.

The structure of this plan is helps an understanding of the code and is documented by its ML type as follows.

```
type component      =   Prim_W    of term      |
                        Comp_W    of string;;

type trace          =   Prod_t    of component list  |
                        Union_t   of int # component;;

typeabbrev witness  =   term # int # trace;;

typeabbrev witnesses =  (string # witness) list;;
```

The following function when given a witness, the predicate it is a witness for, and an object of type *witnesses* will return the theorem that the witness is indeed a witness.

```
let     REX_TAC (witness,depth,tr) witnesses = THENL[
        EXISTS_TAC witness;
        top_beta;
        EXISTS_TAC (nterm int);
        REWRITE_TAC[PRIM_REC_THM];
        TRACE_TAC witnesses tr]
whereTRACE_TAC witnesses = fun
              Prod_t cl     .


where       top_beta    =   CONV_TAC BETA_CONV
and         depth_beta  =   CONV_TAC (DEPTH_CONV BETA_CONV);;
```

3. NON-RECURSIVE EXISTENCE PROOFS

3.1. Primitive Types

3.2. Unions

```
ML
let void_witproof rep_type =
        let [dom; cod] = snd (dest_type rep_type)
 in     "λv:^rep_type.v=(list_mk_pfun [[]:^dom,μx:^cod.T])";;

let witproof ((_, rep_type, terms,_),_,_,results,_) = fun
        P t     .fst (snd (snd (assoc t terms))),TRUTH              |
        T n     .snd (assoc n results),TRUTH            |
        V       .void_witproof rep_type,TRUTH;;

let witproofs g = map (witproof g);;
```

```
ML
let u_witness ((_,u,_,l),_,_,_) t =
        ("(inject ^(dom_val l 1) ^t):^u") ;;
```

```
ML
let uex_tac wit1 thm wit2 = EVERY [
        EXISTS_TAC wit1;
        CONV_TAC BETA_CONV;
        DISJ1_TAC;
        EXISTS_TAC wit2;
        REWRITE_TAC [];
        ACCEPT_TAC thm];;

let uex_thm pred l thm wit2 =
        let goal = ([],"∃x.^pred x")
        and wit1 = "inject ^(dom_val l 1) ^wit2"
 in     TAC_PROOF(goal, uex_tac wit1 thm wit2);;

let do_nonrec_U_ex g (gn,s) =
        let ((_,rep_type,_,n),_,_,d,_) = g
 in     let pred                = snd (assoc gn d)
        and witness,proof  = witproof g (hd s)
 in     (gn, witness, uex_thm pred n proof witness);;
```

3.3. Products

```
ML
let mk_product_ex g gn t = "t",TRUTH;;
```

```
ML
let do_nonrec_P_ex g (gn,s) =
        (gn,
        mk_product_ex g gn (map (witproof g) s));;
```

3.4. Non-recursive types

```
ML
let do_nonrectype_exists ad (name, st) =
        case st of      Union u       .do_nonrec_U_ex ad (name, map snd u)      |
                        Prod p        .do_nonrec_P_ex ad (name, map snd p);;
```

```
ML
let exist_proved u v = true;;

let nonrectypes_estep (g,u,d,r,k) =
        let ok, not_ok = partition ((exist_proved u) o snd) u
        in      ok = [] => failwith ‘nonrectypes_estep - finished‘ |
                        (g,
                        d @ ok,
                        not_ok,
                        r,
                        k @ (map (do_nonrectype_exists (g,u,d,r,k)) ok));;
```

```
ML
let nonrectype_exists (g,u,d,r) =
        careful_repeat_f [‘nonrectypes_estep - finished‘] nonrectypes_estep (g,u,d,r,[]);;
```

4. CONTROL

```
ML
let new_rectypes = make_predicates o do_initiation;;
```

5. THE THEORY 051