Presentation at Cambridge Computer Labs

Roger Bishop Jones

ICL Defence Systems

This is a summary of relevant work for presentation to the hardware verification group at Cambridge Computer Labs. (March 26, 1987)

1. INTRODUCTION

TRANSLATING Z into HOL

LOGICAL FOUNDATIONS

RECURSIVE DATA TYPES in HOL

USING HOL for METAMATHEMATICS

2. Translating Z into HOL

LABELLED PRODUCT DATA TYPES

DISJOINT UNION DATA TYPES

ENUMERATED DATA TYPES

Sum, Lists, Partial Functions

SCHEMA TYPES and CONSISTENCY

3. SUMMARY of OUTPUT on FOUNDATIONS

Logical Foundations and Formal Verification

Implementing Cardelli’s type:type in Turner’s Miranda

The Calculus of Constructions in Miranda

Creative Foundations for Program Verification

A new axiomatisation of the Theory
of Restricted Generality

A Combinatory Theory of Partial Functions

4. Recursive Data Types in HOL

To support recursive definitions of collections
of labelled union and labelled product types.

Work started but not very far progressed.

Uses an ML type for describing data types.

COMPLETED:
computation of underlying representation type

construction of the predicates defining the types

IN PROGRESS:
Construction of existence proofs (painful !)

An awful lot still to be done

5. Using HOL for METAMATHEMATICS

NOTHING DONE YET
- NEEDS RECURSIVE DATA TYPES

SYNTAX = free algebra defined with recursive data types

THEOREMS defined using quotients

QUOTIENT on FREE COMBINATORY ALGEBRA
for models

OBJECTIVE - To PROVE RESULTS
about PROOF THEORETIC STRENGTH

6. Z in HOL

6.1. PRIMITIVES in HOL

CLASS

We assume a fixed lattice of security classifications, elements of which have type *class*.

ML

```
let CLASS = ":*class";;
```

6.2. SCHEMAS in HOL

```
__CLASS_RANGE[p]_____________________
high,low      : CLASS
_________________
high dominates low
_____________________________
```

```
ML
let CLASS_RANGE = new_schema (‘CR‘,[
                        (‘high‘,       ":^CLASS");
                        (‘low‘,        ":^CLASS") ]);;
```

```
ML
new_definition(‘pCLASS_RANGE‘,
        "pCLASS_RANGE (cr:^CLASS_RANGE) =
                dominates (CR_high cr) (CR_low cr)");;
```

6.3. ENUMERATED TYPES in HOL

DOCUMENT_TYPE ≙ Draft | Pre-release | Commentary | Release | Downgrade

ML

```
let DOCUMENT_TYPE = new_enumerated_type
                [       'Draft';
                        'Pre_Release';
                        'Commentary';
                        'Release';
                        'Downgrade'     ];;
```

6.4. Disjoint Unions in HOL

```
OPERATION ≙
        Assign_role<<ASSIGN_ROLE_PARS>>          |
        Remove_role     <<REMOVE_ROLE_PARS>>     |
        Create_doc      <<CREATE_DOC_PARS>>           |
        Create_para     <<CREATE_PARA_PARS>>          |
        Display_doc     <<DISPLAY_DOC_PARS>>          |
        Display_para    <<DISPLAY_PARA_PARS>>     |
        Downgrade_doc   <<DOWNGRADE_DOC_PARS>>        |
        Downgrade_para<<DOWNGRADE_PARA_PARS>>         |
        Edit_para       <<EDIT_PARA_PARS>>            |
        Change_doc_type<<CHANGE_DOC_TYPE_PARS>>
```

ML

```
let OPERATION = new_disjoint_union
[       (‘Assign_role‘,         ASSIGN_ROLE_PARS);
        (‘Remove_role‘,         REMOVE_ROLE_PARS);
        (‘Create_doc‘,          CREATE_DOC_PARS);
        (‘Create_para‘,         CREATE_PARA_PARS);
        (‘Display_doc‘,         DISPLAY_DOC_PARS);
        (‘Display_para‘,        DISPLAY_PARA_PARS);
        (‘Downgrade_doc‘,       DOWNGRADE_DOC_PARS);
        (‘Downgrade_para‘,      DOWNGRADE_PARA_PARS);
        (‘Edit_para‘,           EDIT_PARA_PARS);
        (‘Change_doc_type‘,     CHANGE_DOC_TYPE_PARS)       ];;
```

6.5. FUNCTIONS over DISJOINT UNIONS

```
do_op       : STATE_STEP_FUNCTION
_____________________

do_op (Assign_role pars) s          = assign_role pars s
do_op (Remove_role pars) s          = remove_role pars s
do_op (Create_doc pars) s           = create_doc pars s
do_op (Create_para pars) s          = create_para pars s
do_op (Display_doc pars) s          = display_doc pars s
do_op (Display_para pars) s         = display_para pars s
do_op (Downgrade_doc pars) s        = downgrade_doc pars s
do_op (Downgrade_para pars) s       = downgrade_para pars s
do_op (Edit_para pars) s        = edit_para pars s
do_op (Change_doc_type pars) s      = change_doc_type pars s
```

ML

```
dju_case_def ‘do_op‘ OPERATION [
(‘Assign_role‘,          "assign_role       :^ASSIGN_ROLE");
(‘Remove_role‘,          "remove_role       :^REMOVE_ROLE");
(‘Create_doc‘,           "create_doc        :^CREATE_DOC");
(‘Create_para‘,          "create_para       :^CREATE_PARA");
(‘Display_doc‘,          "display_doc       :^DISPLAY_DOC");
(‘Display_para‘,         "display_para      :^DISPLAY_PARA");
(‘Downgrade_doc‘,        "downgrade_doc     :^DOWNGRADE_DOC");
(‘Downgrade_para‘,       "downgrade_para    :^DOWNGRADE_PARA");
(‘Edit_para‘,            "edit_para         :^EDIT_PARA");
(‘Change_doc_type‘,      "change_doc_type   :^CHANGE_DOC_TYPE")
                                        ];;
```

6.6. Z SEMANTICS and PROOF THEORY

Are empty types legal? e.g.:

INT_PAIR[p]

left, right :INT

---

left > right ∧ right > left

If not, how do we know that our schemas are legal?

If so, what is the status of declarations using empty types? e.g.:

ip:INT_PAIR

Can we then assert:

? ip.left > ip.right ∧ ip.right > ip.left

?

There is a solution which provides a consistent proof theory for Z, and which translates into HOL.

We interpret schemas with non-empty bodies in the following way:

$$\begin{aligned}&\mathrm{INT\_PAIR} = \\ &\quad \{[\mathrm{left,right:INT}] \mid \mathrm{left} > \mathrm{right} \land \mathrm{right} > \mathrm{left}\} \\ &\quad \cup \{\mu \mathrm{p}:[\mathrm{left,right:INT}] \mid \mathrm{left} > \mathrm{right} \land \mathrm{right} > \mathrm{left}\}\end{aligned}$$

Inference rules relating to schema types must be qualified.

$$\begin{aligned}&(\exists \mathrm{x}:[\mathrm{left,right:INT}].\mathrm{left} > \mathrm{right} \land \mathrm{right} > \mathrm{left}) \\ &\Rightarrow \forall \mathrm{ip:INT\_PAIR}. \\ &\quad \mathrm{ip.left} > \mathrm{ip.right} \land \mathrm{ip.right} > \mathrm{ip.left}\end{aligned}$$

7. RECURSIVE DATA TYPES in HOL

7.1. ML type of type descriptions

```
ML
type field_type      =       V | T of string | P of type;;

typeabbrev field_desc     = string # field_type;;

type struc_type  = Prod          of field_desc list   |
                   Union         of field_desc list;;

typeabbrev rec_types = (string # struc_type) list;;
```

7.2. Sample Type Description

ML

```
let     UNION       = (‘UNION‘,
        Union [‘A‘,V ;‘B‘,V ;‘C‘,P ":bool";
                ‘D‘, P ":bool→bool"])

and     APP   = (‘APP‘,
        Prod [‘fun‘, T ‘COMB‘; ‘arg‘, T ‘COMB‘])

and     COMB        = (‘COMB‘,
        Union [‘K‘, V; ‘S‘, V; ‘app‘, T ‘APP‘]);;

let name = ‘COMBINATOR‘;;

let rec_types = [APP; COMB; UNION];;

let nrt = new_rectypes (name, rec_types);;
```

7.3. Sample Predicates

```
nrt =
(‘COMBINATOR‘,
":((void + (void + (void + void)))list,bool + (bool → bool))pfun",
[":bool",
 "λv. INL v",
 "mk_is_primitive(λv. INL v)",
 "(λv. INL v)(μx. T)";
 ":bool → bool",
 "λv. INR v",
 "mk_is_primitive(λv. INR v)",
 "(λv. INR v)(μx. T)"],
4),
[],

[‘APP‘,
 Prod [‘fun‘,T ‘COMB‘; ‘arg‘,T ‘COMB‘];
 ‘COMB‘,
 Union [‘K‘,V; ‘S‘,V; ‘app‘,T ‘APP‘];
 ‘UNION‘,
 Union [‘A‘,V; ‘B‘,V; ‘C‘,P ":bool"; ‘D‘,P ":bool → bool"]],
```

```
['UNION',
 "λv.
   (∃c.
     (λv. v = list_mk_pfun[[],(μx. T)])c ∧ (v = inject(INL(μx. T))c))
   (∃c.
     (λv. v = list_mk_pfun[[],(μx. T)])c ∧
     (v = inject(INR(INL(μx. T)))c))
   (∃c.
     mk_is_primitive(λv. INL v)c ∧ (v =
                                    inject(INR(INR(INL(μx. T))))c))
   (∃c.
     mk_is_primitive(λv. INR v)c ∧ (v =
                                    inject(INR(INR(INR(μx. T))))c))";
```

```
‘APP‘,
"λr.
 ∃n.
  (λx. FST x)
  (PRIM_REC
   (λr. F,F)
   (λf m v.
    (∃c’ c.
     ((λx. (λx. SND x)(f x))c’ ∧ (λx. (λx. SND x)(f x))c) ∧
     (s = mk_schema[INL(μx. T),c’;INR(INL(μx. T)),c])),
    ((∃c.
      (λv. v = list_mk_pfun[[],(μx. T)])c ∧
      (v = inject(INL(μx. T))c))
     (∃c.
      (λv. v = list_mk_pfun[[],(μx. T)])c ∧
      (v = inject(INR(INL(μx. T)))c))
     (∃c.
      (λx. (λx. FST x)(f x))c ∧ (v = inject(INR(INR(INL(μx. T))))c))))
  n
  r)";
```

```
‘COMB‘,
"λr.
 ∃n.
  (λx. SND x)
  (PRIM_REC
   (λr. F,F)
   (λf m v.
    (∃c’ c.
     ((λx. (λx. SND x)(f x))c’ ∧ (λx. (λx. SND x)(f x))c) ∧
     (s = mk_schema[INL(μx. T),c’;INR(INL(μx. T)),c])),
    ((∃c.
      (λv. v = list_mk_pfun[[],(μx. T)])c ∧
      (v = inject(INL(μx. T))c))
     (∃c.
      (λv. v = list_mk_pfun[[],(μx. T)])c ∧
      (v = inject(INR(INL(μx. T)))c))
     (∃c.
      (λx. (λx. FST x)(f x))c ∧ (v = inject(INR(INR(INL(μx. T))))c))))
   n
   r)"]
```