Universal Set Theory in SML (document 45)

Roger Bishop Jones

ICL Defence Systems

ABSTRACT

This document consists of a formal specification in SML of a variant of classical first order set theory. This particular variant consists of ZFC augmented by a hierarchy of universes, each enjoying the same closure properties as ZFC. To cap it all we have classes.

This issue is update by removing an error in the definition of substitution and by modification and extension to the auxiliary functions, some of which will be used in subsequent documents. Use of the SML 'modules' has been introduced.

Changes Forecast

Future changes may be made to make the theory practicable for use in deriving the proof theory of langauges whose semantics is defined in terms of UST. This may involve introducing some "urelements" rather than sticking to a pure set theory, since the terms needed to give denotations to primitive types such as natural numbers rapidly get out of hand.

1. INTRODUCTION

This document provides a formal description of a variant of classical set theory axiomatised in first order logic. This is claimed to be a suitable formal foundation for the VDM specification language. It is richer than strictly necessary, but this makes some of the constructions simpler, and provides more flexibility for dealing with problems in the current model.

It could be used simply to give a formal description of the current type model, which would help in deriving the proof theory. In this case the iterative constructions in the model could be simplified by taking intersections of sets containing the basic types and closed under the type constructors.

More radical changes are also possible, such as closing the type universe under a full partial function space constructor, and eliminating the two tier construction of the present model. Special provision is made in the choice function to smooth the use of graphs to represent partial functions over spaces not containing a bottom element.

Even after such constructions there remain larger sets which will support semantics for polymorphism and modularity.

The set theory is roughly NBG with added universes. The smallest universe is the universe of ZFC, and every universe is a member of a larger universe with similar closure properties.

The specification is written in the language SML (standard ML).

SML is described in ECS-LFCS-86-2 (University of Edinburgh, Laboratory for Foundations of Computer Science).

2. ABSTRACT SYNTAX

```
infix 4 ⇔;
infix 5 ☞;
infix 6 ;
infix 7 ∧;
infix 8 == ∈ ε -- ⊆ t_subs_for subs_for;
infix 9 inn;
structure UST =
struct
structure FTM =
struct
```

```
type var      = string;
```

```
datatype
      formula     =     ¬ of formula                    |
                        op ☞ of (formula * formula)     |
                        ∀ of (var * formula)            |
                        op == of (term * term)          |
                        op ∈ of (term * term)
```

These constructs have their normal meaning, viz. negation, material implication, existential quantification, equality and membership respectively.

```
and   term  =           Var of var                      |
                        μ of term                       |
                        comp of (var * formula)         |
                        V of term;
```

The meanings of these term constructors are:

Var Variables

μ choice function (following Oxford Z usage) yields a set, which will be a member of its argument except in the case that the argument is ∅.

comp
comprehension, "comp(a,b)" is more usually written {a | b}
comprehension rather than separation is allowed but does not necessarily yield a set. It yields the class of sets which satisfy the predicate "λ a.b".

V V x is a "universe" containing x
A universe is a set which satisfies very liberal closure conditions, it is closed under all the set forming operations of ZF. Every set is a member of some universe, and V x is a universe containing x.

3. SUBSTITUTION

3.1. Set Theory in the Metalanguage (sml)

First a few definitions which enable us to talk about lists as if they were sets.

```
fun   x ε [] = false                          |
      x ε (y::z)    = x = y orelse x ε z;
```

```
fun   m∪ []= []                               |
      m∪ (a::b)   = a @ (m∪ b);
fun   [] -- a       = []                              |
      (h::t) -- a   = if h=a then t -- a else h::(t -- a);
```

Note here that:

m∪ is here defined as a single prefix operator meaning distributed union.

@ is the concatenation operator built into SML.

3.2. Free Variables

We define the free variables in a formula or term as follows:

```
fun   term_freevars (Var v)           = [v]                                   |
      term_freevars (μ t)       = term_freevars t                       |
      term_freevars (comp (v,f))      = freevars f -- v                       |
      term_freevars (V t)             = term_freevars t

and   term_list_freevars term_list    = m∪ (map term_freevars term_list)
```

```
and   freevars (¬ f)              = freevars f                              |
      freevars (f1 ☞ f2)          = freevars f1 @ (freevars f2)             |
      freevars (∀ (v,f))          = freevars f -- v                         |
      freevars (t1 == t2)         = term_freevars t1 @ (term_freevars t2)   |
      freevars (t1 ∈ t2)          = term_freevars t1 @ (term_freevars t2);
```

3.3. Substitutions

First define substitutions into terms.

```
fun   primed nv varl =
                  if    nv ε varl
                  then  primed (nv^"'") varl
                  else  nv;
fun   freev v terml = primed v (term_list_freevars terml);
```

```
datatype ('a,'b)inc = op inn of ('a * 'b);
```

```
fun   term t_subs_for ivar inn (Var v)
          =       if    ivar = v
                  then  term
                  else  Var v                                               |
      term t_subs_for ivar inn (μ t)
          =       (μ (term t_subs_for ivar inn t))                          |
      term t_subs_for ivar inn (comp(v,f))
          =       if    ivar = v
                  then  (comp(v,f))
                  else  let   val nv = primed v (term_freevars term @
                                                 (freevars f -- v))
                              val nf = term subs_for ivar inn
                                       ((Var nv) subs_for v inn f)
                        in    (comp(nv,nf))
                        end                                                 |
      term t_subs_for ivar inn (V t)
          =       V (term t_subs_for ivar inn t)
and   curried_t_subs term ivar term'
          =       term t_subs_for ivar inn term'
```

Next substitutions into formulae.

```
and   term subs_for ivar inn (¬ form)
            =     ¬ (term subs_for ivar inn form)                    |
      term subs_for ivar inn (form1 ☞ form2)
            =     (term subs_for ivar inn form1)
                  ☞ (term subs_for ivar inn form2)                   |
      term subs_for ivar inn (∀ (ivar',form))
            =     if    ivar = ivar'
                  then  (∀ (ivar',form))
                  else  let   val nv = primed ivar' (term_freevars term @
                                                     (freevars form -- ivar'))
                              val nf = term subs_for ivar inn
                                          ((Var nv) subs_for ivar' inn form)
                        in    (∀(nv,nf))
                        end                                          |
      term subs_for ivar inn (t1 == t2)
            =     (term t_subs_for ivar inn t1)
                  == (term t_subs_for ivar inn t2)                   |
      term subs_for ivar inn (t1 ∈ t2)
            =     (term t_subs_for ivar inn t1)
                  ∈ (term t_subs_for ivar inn t2)
and   curried_subs a b c = a subs_for b inn c;
```

4. DERIVED CONSTRUCTORS

4.1. Boolean Connectives

```
fun a  b      = (¬ a) ☞ b;
fun a ∧ b     = ¬( (¬ a)  (¬ b));
fun a ⇔ b     = (a ☞ b) ∧ (b ☞ a);
```

4.2. Existence and Subset

```
fun ∃ (v,f)   = ¬ (∀(v,¬ f));
fun a ⊆ b     =     let   val x = freev "x" [a,b]
                          in    ∀ (x,(Var x ∈ a) ☞ (Var x ∈ b))
                          end;
```

4.3. Sets

This set theory has classes. Everything is a class, a set is a class which is the member of a class (or a set).

```
fun set a =   let val x = (freev "x" [a])
              in      ∃(x, a ∈ (Var x))
              end;
```

The empty set is the class which has no members (an axiom asserts that this is a set).

```
val ∅ = comp("x",¬(Var "x" == Var "x"));
```

4.4. Pairs Ordered Pairs and unit sets.

The following definitions of pair and ordered pair (opair) are the usual ones.

```
fun pair a b =
              let val x = (freev "x" [a,b])
              in      comp (x,(Var x==a)  (Var x==b))
              end;

fun opair a b = pair a (pair a b);

fun left a =
              let     val x = (freev "x" [a])
                      and y = (freev "y" [a])
              in      μ (comp(x, ∃(y, a == opair (Var x) (Var y))))
              end;

fun right a =
              let     val x = (freev "x" [a])
                      and y = (freev "y" [a])
              in      μ (comp(x, ∃(y, a == opair (Var y) (Var x))))
              end;

fun unit a = pair a a;
```

4.5. Power Set, Union, Intersection

```
fun ℙ a =   let val x = (freev "x" [a])
            in    comp (x,(Var x ⊆ a))
            end;

fun ∪ a =
            let   val x = (freev "x" [a]);
                  val y = (freev "y" [a])
            in    comp (x,∃ (y,(Var x ∈ (Var y)) ∧ (Var y ∈ a)))
            end;

fun ∩ a =
            let   val x = (freev "x" [a]);
                  val y = (freev "y" [a])
            in    comp (x,∀ (y,(Var y ∈ a) ☞ (Var x ∈ (Var y))))
            end;
```

4.6. Relations and (Partial) Functions

A relation is simply a set of ordered pairs.

```
fun is_rel a =      let   val w = freev "w" [a]
                    and x = freev "x" [a]
                    and y = freev "y" [a]
                    and z = freev "z" [a]
                    in    ∀(w, (Var w ∈ a) ☞
                              ∃(x,∃(y, ((Var w) == opair (Var x) (Var y)))))
                    end;
```

dom and ran are respectively the domain and range of a relation.

```
fun dom f =
            let   val x = freev "x" [f]
                  and y = freev "y" [f]
            in    comp (x,∃(y,opair (Var x) (Var y) ∈ f))
            end;
fun ran f = let   val x = freev "x" [f]
                  and y = freev "y" [f]
            in    comp (x,∃(y,opair (Var y) (Var x) ∈ f))
            end;
```

The following defines the property of being a single valued relation (many-one).

```
fun is_sv a =       let     val x = freev "x" [a]
                            and y = freev "y" [a]
                            and z = freev "z" [a]
                    in      is_rel a ∧
                    ∀(x,∀(y,∀(z,
                    (opair (Var x) (Var y) ∈ a) ∧
                    (opair (Var x) (Var z) ∈ a) ☞ ((Var y) == (Var z)))))
                end;
```

The following function is the domain restriction of a relation. The domain restriction of r to d is the class of ordered pairs in r whose left member is in d.

```
fun dom_restr r d =
    let     val x = freev "x" [r,d]
            and y = freev "y" [r,d]
    in      comp(x, (Var x) ∈ r ∧ (left (Var x) ∈ d))
    end;
```

Relational override is defined by a domain restriction and a union.

```
fun rel_over r1 r2 = ∪ (pair (dom_restr r1 (dom r2)) r2);
```

Relational update is a version of override for a single value.

```
fun rel_update r1 a v = rel_over r1 (opair a v);
```

5. THE ABSTRACT DATA TYPE THEOREM

The set theory is formalised as a hilbert style axiom system by defining an abstract data type of theorems, where theorems are represented by formulae.

5.1. Inference Rules

Inference rules are formalised as functions from theorems to theorems, and axioms schemata as functions on any type yeilding theorems.

```
end; (* of FTM *)
structure THM = struct
local open FTM in
abstype theorem =  of formula
with
```

5.1.1. Modus Ponens

Modus Ponens takes two theorems of the form A, A ☞ B and returns
B. If the second theorem is not of the correct form then it will just return the first. This preserves the soundness of the logic while avoiding the extra complexity of exception handling.

```
fun   MP ( A) ( (op ☞ (B,C))) = if A=B then   C else   A                |
      MP x y                  = x;
```

5.1.2. Generalisation

UI takes as parameters a theorem and a variable name, returning a generalisation of the theorem.

```
fun UI ( A) x =  (∀(x,A));
```

5.2. Propositional Axioms

These propositional axiom schemata are parameterised by arbitrary formulae.

```
fun P1 A B       =  ((A ☞ B) ☞ A);
fun P2 A B C     =  ((A ☞ B ☞ C) ☞ (A ☞ B) ☞ (A ☞ C));
fun P3 A B       =  ((¬ A ☞ ¬ B) ☞ (B ☞ A));
```

5.3. Axioms of Quantification

In the quantification schemata A and B are arbitrary formulae, x is a variable name and t a term. Instead of rejecting combinations which would not be sound variables will be renamed (by priming) as appropriate to arrive at a valid instance.

```
fun Q1 A x t     =  (∀(x,A) ☞ (t subs_for x inn A));
fun Q2 A x       =  (A ☞ ∀(primed x (freevars A),A));
fun Q3 A B x     =  (∀(x,A ☞ B) ☞ ∀(x,A) ☞ ∀(x,B));
```

5.4. Equality and Membership

Equality need not be primitive, axiom EQ could be replaced by a similar definition in the meta-language.

```
fun EQ a b x      =
                  let   val nx = freev x [a,b]
                  in    ((a==b) ⇔ (∀(nx,
                              (Var nx ∈ a ⇔ Var nx ∈ b))))
                  end;
fun EXT a b x     =
                  let   val nx = freev x [a,b]
                  in    ((a==b) ⇔ (∀(nx,
                              (a ∈ (Var nx) ⇔ b ∈ (Var nx)))))
                  end;
```

5.5. Comprehension

By contrast with Zermelo-Fraenkel we have comprehension rather than separation. This is sound because a comprehension does not always yield a set, it may yield a class, and because, as the following axiom states, it yields a class which contains just those sets which satisfy the predicate. Classes are never members of anything, hence the class of all sets which are not members of themselves is not a member of itself.

```
fun COM x p       =  (∀ (x,set (Var x) ∧ p ⇔ (Var x) ∈ comp(x,p)));
```

5.6. Sets

The empty class is a set, as is the pair formed from any two sets. This latter axiom may be redundant.

```
fun S∅ x          =  (set ∅);
fun Spair a b     =  (set a ∧ set b ☞ set (pair a b));
```

5.7. Universes and Closure

The function U maps each set onto a universe containing it. A universe is a transitive set (every member of it is a subset of it) closed under the set forming operations of Zermelo-Fraenkel, viz power set formation, union, and replacement. These axioms imply also closure under separation, pairing, and choice from non-empty sets. The availability of classes makes the statement of the replacement axiom smoother.

Urep looks slightly stronger than the equivalent closure condition in Cohn, since it asserts that the range of the function is in the same universe as the domain, Cohn merely states that the union of the range is in the universe and that the range itself is a set.

```
fun Umem u      = (set u ☞ u ∈ (V u) ∧ set (V u));
fun Utrans u x  = (x ∈ (V u) ☞ x ⊆ (V u));
fun Upower u a  = (a ∈ (V u) ☞ (ℙ a ∈ (V u)));
fun Union u a   = (a ∈ (V u) ☞ (∪ a ∈ (V u)));
fun Urep u f=  (is_sv f ∧ (dom f) ∈ (V u) ∧ (ran f) ⊆ (V u)
                    ☞      (ran f) ∈ (V u));
```

5.8. Choice

The choice function maps every non-empty class onto a member of itself. It maps the empty set onto an unspecified class.

This formulation of the choice axiom is in the first clause like the usual formulation for ZF. The second clause is my innovation, in order to ensure that the value obtained on application of a partial function to an element outside its domain is distinct from any element in its range. (it is μ ∅, which is by this axiom a proper class and hence may not be in the range of any function).

```
fun CH t =  ((set (μ t) ⇔ (¬ (t == ∅)))
                    ∧ ((¬ (t == ∅)) ⇔ (μ t) ∈ t));
```

5.9. Foundation

The axiom of foundation is needed to permit inductive definitions.

```
fun FO t =  let     val y = primed "y" (term_freevars t)
            in      (t==∅
                    ∃(y,Var y ∈ t ∧ (∩ (pair t (Var y)) == ∅)))
            end;
```

5.10. Extracting the Formula from a Theorem

The following procedure allows inspection of the content of a theorem from outside the abstract data type.

```
fun formula( t) = t;
```

5.11. End of Abstract Data Type

```
end; (* of local clause *)
end; (* of abstract data type *)
end; (* of structure THM *)
end; (* of structure UST *)
open UST.FTM UST.THM;
```

The types inferred by the SML compiler were:

```
> type theorem
  type var = string
  datatype term = V of term | Var of var | μ of term | comp of var * formula
  datatype ('a,'b) inc = inn of 'a * 'b
  datatype formula = ∀ of var * formula | == of term * term | ☞ of formula *
 formula | ∈ of term * term | ¬ of formula
  val Utrans = fn : term → (term → theorem)
  val Urep = fn : term → (term → theorem)
  val Upower = fn : term → (term → theorem)
  val Union = fn : term → (term → theorem)
  val Umem = fn : term → theorem
  val UI = fn : theorem → (var → theorem)
  val Spair = fn : term → (term → theorem)
  val S∅ = fn : 'a → theorem
  val Q3 = fn : formula → (formula → (var → theorem))
  val Q2 = fn : formula → (string → theorem)
  val Q1 = fn : formula → (var → (term → theorem))
  val P3 = fn : formula → (formula → theorem)
  val P2 = fn : formula → (formula → (formula → theorem))
  val P1 = fn : formula → (formula → theorem)
  val MP = fn : theorem → (theorem → theorem)
  val FO = fn : term → theorem
  val EXT = fn : term → (term → (string → theorem))
  val EQ = fn : term → (term → (string → theorem))
  val COM = fn : var → (formula → theorem)
  val CH = fn : term → theorem
  con ¬ = fn : formula → formula
  val unit = fn : term → term
  val term_list_freevars = fn : (term list) → (var list)
  val term_freevars = fn : term → (var list)
  val t_subs_for = fn : (term * ((var,term) inc)) → term
  val subs_for = fn : (term * ((var,formula) inc)) → formula
  val set = fn : term → formula
  val right = fn : term → term
  val rel_update = fn : term → (term → (term → term))
  val rel_over = fn : term → (term → term)
  val ran = fn : term → term
  val primed = fn : string → ((string list) → string)
  val pair = fn : term → (term → term)
  val opair = fn : term → (term → term)
  val m∪ = fn : (('a list) list) → ('a list)
  val left = fn : term → term
  val is_sv = fn : term → formula
  val is_rel = fn : term → formula
  con inn = fn : ('a * 'b) → (('a,'b) inc)
```

```
val freevars = fn : formula → (var list)
val freev = fn : string → ((term list) → string)
val dom_restr = fn : term → (term → term)
val dom = fn : term → term
val curried_t_subs = fn : term → (var → (term → term))
val curried_subs = fn : term → (var → (formula → formula))
con comp = fn : (var * formula) → term
val  = fn : (formula * formula) → formula
con ∈ = fn : (term * term) → formula
val ⊆ = fn : (term * term) → formula
val ∅ = comp ("x",¬(== (Var "x",Var "x"))) : term
val ∪ = fn : term → term
val ∩ = fn : term → term
con μ = fn : term → term
val ε = fn : (''a * (''a list)) → bool
val ℙ = fn : term → term
val ∃ = fn : (var * formula) → formula
con Var = fn : var → term
con V = fn : term → term
con ☞ = fn : (formula * formula) → formula
con == = fn : (term * term) → formula
val ⇔ = fn : (formula * formula) → formula
val ∧ = fn : (formula * formula) → formula
val -- = fn : (('a list) * ''a) → (''a list)
con ∀ = fn : (var * formula) → formula
```