Universal Set Theory mk 2

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. Proper classes are also available.

Changes in this issue

This version of UST takes several steps in the direction of usability. Various logical operators have been included as primitive which were previously derived, propositional variables have been introduced, and special syntax for integers and strings.

All pure tautologies are now axioms. Substitution rules have been included for for propositional variables, for equivalence theorems and equality theorems.

Changes Forecast

A very crude local structuring mechanism may be included. The axiom of well foundation may be replaced by an equivalent axiom of noetherian induction. The choice function may be dropped in favour of an axiom of choice not using the choice function.

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

Here we define the priorities of the infix operators introduced.

```sml
infix 4 ⇔;
infix 5 ⊈;
infix 6 ;
infix 7 ∧;
infix 8 == ∈ ⊆
infix 8 subs_for bl_subs_for psubs_for bl_psubs_for
t_subs_for t_bl_subs_for t_psubs_for t_bl_psubs_for;
infix 9 inn;
```

This document defines one structure, UST (Universal Set Theory), containing several sub-structures. The first, FTM (Formula and TerM) defines the syntax of the language.

```sml
structure UST =
struct
structure FTM =
struct

type var = string;
```
datatype formula =
  Pvar of string                           (* propositional variable *)
  ¬ of formula                            (* logical negation *)
  op $\&$ of (formula * formula)          (* material implication *)
  op $\lor$ of (formula * formula)         (* logical or *)
  op $\land$ of (formula * formula)        (* logical and *)
  op $\leftrightarrow$ of (formula * formula) (* logical equivalence *)
  $\forall$ of (var * formula)            (* universal quantification *)
  $\exists$ of (var * formula)             (* existential quantification *)
  op $==$ of (term * term)                 (* equality *)
  op $\in$ of (term * term)                (* membership *)
  T                                        (* truth *)

These constructs have their normal meanings.

and term =
  Var of var                               (* individual variable *)
  $\mu$ of term                            (* choice function *)
  comp of (var * formula)                  (* set comprehension *)
  V of term                                (* Universe *)
  Pair of (term * term)                    (* unordered pairs *)
  $\mathbf{P}$ of term                      (* power set *)
  $\cup$ of term                           (* distributed union *)
  $\cap$ of term                           (* distributed union *)
  Nat of int                               (* Natural numbers *)
  Lit of string                            (* strings *)
  $\emptyset$;                              (* )

val F = $\neg$ T; (* false *)
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", and may be a proper class.

**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 ZFC. Every set is a member of some universe, and V x is a universe containing x.

**Pair**  
Pair (x,y) is the set containing just x and y, normally written \{x,y\}.

**P**  
The Power set constructor, yielding the set of all subsets of some set.  
e.g. P\{1,2\} = \{\{\},\{1\},\{2\},\{1,2\}\}

**∪**  
∪ is distributed union, the union of a set of sets. e.g. \bigcup\{\{1,2\},\{2,3\}\} = \{1,2,3\}

**∩**  
∩ is distributed intersection, the intersection of a set of sets. e.g. \bigcap\{\{1,2\},\{2,3\}\} = \{2\}

**Nat**  
These are the natural numbers including zero. Note that "Nat - i = Nat i". This is just a convenient shorthand for the Von-Neumann representation for the natural numbers, they are not 'urelements'.

**Lit**  
Strings are also provided with abbreviations.

**∅**  
The empty set.

### 3. PRELIMINARIES

#### 3.1. Set Theory in the Metalanguage (sml)

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

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

fun m∪ [] = []
     | m∪ (a::b) = a @ (m∪ b);
```

Note here that:

- m∪ is here defined as a single prefix operator meaning distributed union.
- @ is the concatenation operator built into SML.
fun [] -- a = [] |
    (h::t) -- a = if h=a then t -- a else h::(t -- a);

For many purposes the duplication which may arise in the use of these operators is unimportant, for some however duplicates must be removed.

fun pack_set [] = [] |
    pack_set (h::t) = (h:: (pack_set (t -- h)));

3.2. Free Variables

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

fun term_freevars (Var v) = [v] |
fun term_freevars (μ t) = term_freevars t |
fun term_freevars (comp (v,f)) = freevars f -- v |
fun term_freevars (V t) = term_freevars t |
fun term_freevars (Pair (f,s)) = term_freevars f @ (term_freevars s) |
fun term_freevars (P t) = term_freevars t |
fun term_freevars (∪ t) = term_freevars t |
fun term_freevars (∩ t) = term_freevars t |
fun term_freevars (Nat i) = [] |
fun term_freevars (Lit s) = [] |
fun term_freevars ∅ = [] |
and term_list_freevars term_list = m ∪ (map term_freevars term_list) |

and freevars (Pvar p) = [] |
    freevars (¬ f) = freevars f |
    freevars (f1 @ f2) = freevars f1 @ (freevars f2) |
    freevars (f1 @ f2) = freevars f1 @ (freevars f2) |
    freevars (f1 @ f2) = freevars f1 @ (freevars f2) |
    freevars (f1 @ f2) = freevars f1 @ (freevars f2) |
    freevars (f1 @ f2) = freevars f1 @ (freevars f2) |
    freevars (f1 @ f2) = freevars f1 @ (freevars f2) |
    freevars (f1 @ f2) = freevars f1 @ (freevars f2) |
    freevars (f1 @ f2) = freevars f1 @ (freevars f2) |
    freevars (f1 == t2) = term_freevars t1 @ (term_freevars t2) |
    freevars (t1 ∈ t2) = term_freevars t1 @ (term_freevars t2) |
    freevars T = [];
3.3. Propositional Variables

A pure propositional formula is one which contains no terms or quantification. The following procedure will return the set of propositional variables in a pure propositional formula. In the case that the formula is not pure, occurrences of proposition variables inside quantifiers or inside terms will be ignored.

\[
\text{fun propvars (Pvar p) } = \{ p \} |
\]
\[
\text{propvars (\neg f) } = \text{propvars f} |
\]
\[
\text{propvars (f1 \#& f2) } = \text{propvars f1 @ (propvars f2)} |
\]
\[
\text{propvars (f1 \& f2) } = \text{propvars f1 @ (propvars f2)} |
\]
\[
\text{propvars (f1 \iff f2) } = \text{propvars f1 @ (propvars f2)} |
\]
\[
\text{propvars f } = [];
\]
3.3.1. Simplification of formulae

The following procedure simplifies a formula while assigning a truth value to a propositional variable. The parameter v is the variable to which assignment is being made, t is the truth value being assigned, and the third parameter is the formula to be simplified.

```
fun simpl v t (Pvar p) = if p = v then t else (Pvar p) |
  simpl v t (¬ f) =
    (case (simpl v t f) |
      of (¬ f') => f' | f' => (¬ f')) | |
  simpl v t (f1 ⇔ f2) =
    (case (simpl v t f1, simpl v t f2) |
      of (¬ T, x) => T |
       (x, T) => T |
       (T, ¬ T) => ¬ T |
       (x, y) => x ⇔ y) | |
  simpl v t (f1 ∧ f2) =
    (case (simpl v t f1, simpl v t f2) |
      of (T, T) => T |
       (¬ T, x) => F |
       (x, ¬ T) => F |
       (x, y) => x ∧ y) | |
  simpl v t (f1 ⇔ f2) =
    (case (simpl v t f1, simpl v t f2) |
      of (T, T) => T |
       (¬ T, ¬ T) => T |
       (T, ¬ T) => F |
       (¬ T, T) => F |
       (x, y) => x ⇔ y) | |
  simpl v t f = f;
```
4. SUBSTITUTION

This section contains definitions of various substitution operations needed in defining the axioms and inference rules which follow later. These concern the substitution of formulae for propositional variables, or of terms for individual variables (variables ranging over the domain of discourse), in some term or formula. For each of the above four cases functions are provided which rename bound variables which would otherwise capture a free variable in the term or formula being substituted, and similar functions are provided which perform no variable renaming, allowing free variables to be captured. These latter functions are used in defining the inference rules allowing substitution of equals for equals or equivalents for equivalents, which in neither case need consider whether variables free in the equality or equivalence statement are also free in the premiss or conclusion of the inference rule.

The procedures for substituting into terms and formulae will in each case be mutually recursive, following the mutual recursion in the abstract syntax.

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

Function 'primed' may be used to find new variable names which do not clash with the variables supplied as the second argument. The first argument will have primes appended to it until it is distinct from all the names in the second argument.

fun primed nv varl =
    if nv \in varl
    then primed (nv"''") varl
    else nv;

fun freev terml v = primed v (term_list_freevars terml);
```
4.1. Substitution of Terms

4.1.1. Into Terms

First substitution of a term (term) for a free variable (ivar) in a term is defined. Renaming of bound variables will be accomplished if this is necessary to prevent capture of the free variables of the term to be substituted.

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)
   term t_subs_for ivar inn (Pair (f,s))
   = Pair ( term t_subs_for ivar inn f,
              term t_subs_for ivar inn s )
   term t_subs_for ivar inn (Π t)
   = Π (term t_subs_for ivar inn t)
   term t_subs_for ivar inn (∪ t)
   = ∪ (term t_subs_for ivar inn t)
   term t_subs_for ivar inn (∩ t)
   = ∩ (term t_subs_for ivar inn t)
   term t_subs_for ivar inn (Nat i)
   = Nat i
   term t_subs_for ivar inn (Lit s)
   = Lit s
   term t_subs_for ivar inn ∅
   = ∅

and curried_t_subs term ivar term’
   = term t_subs_for ivar term’

DBC/FMU/RBJ/116 Issue 0.6 UST mk 2 Page 9
4.1.2. Into Formulae

Next substitution of a term for a free variable in a formula.

\[
\begin{align*}
\text{and} \quad \text{term subs for ivar inn (Pvar p)} & = (Pvar p) & \\
\text{term subs for ivar inn (\neg form)} & = \neg (\text{term subs for ivar inn form}) & \\
\text{term subs for ivar inn (form1 \land form2)} & = (\text{term subs for ivar inn form1}) \land (\text{term subs for ivar inn form2}) & \\
\text{term subs for ivar inn (form1 \leftrightarrow form2)} & = (\text{term subs for ivar inn form1}) \leftrightarrow (\text{term subs for ivar inn form2}) & \\
\text{term subs for ivar inn (\forall (ivar',form))} & = \text{if} \quad \text{ivar = ivar'} & \\
& \quad \text{then} \quad (\forall (ivar',form)) & \\
& \quad \text{else} \quad \text{let} \quad \text{val nv = primed ivar'} & \\
& \quad \text{term_freevars term} & \\
& \quad (\text{freevars form} \rightarrow \text{ivar'}) & \\
& \quad \text{val nf = term subs for ivar inn} & \\
& \quad ((\text{Var nv}) \text{ subs_for ivar'} \text{ inn form}) & \\
& \quad \text{in} \quad (\forall (nv,nf)) & \\
& \quad \text{end} & \\
\text{term subs for ivar inn (\exists (ivar',form))} & = \text{if} \quad \text{ivar = ivar'} & \\
& \quad \text{then} \quad (\exists (ivar',form)) & \\
& \quad \text{else} \quad \text{let} \quad \text{val nv = primed ivar'} & \\
& \quad \text{term_freevars term} & \\
& \quad (\text{freevars form} \rightarrow \text{ivar'}) & \\
& \quad \text{val nf = term subs for ivar inn} & \\
& \quad ((\text{Var nv}) \text{ subs_for ivar'} \text{ inn form}) & \\
& \quad \text{in} \quad (\exists (nv,nf)) & \\
& \quad \text{end} & \\
\text{term subs for ivar inn (t1 == t2)} & = (\text{term t_subs for ivar inn t1}) == (\text{term t_subs for ivar inn t2}) & \\
\text{term subs for ivar inn (t1 \in t2)} & = (\text{term t_subs for ivar inn t1}) \in (\text{term t_subs for ivar inn t2}) & \\
\text{term subs for ivar inn T = T} & = & \\
\text{and} \quad \text{curried_subs a b c = a subs_for b inn c;} & = & \\
\end{align*}
\]
4.2. Blind Substitution of Terms

These variants ignore capture of variables.

4.2.1. Into Terms

First define substitution of a term for a free variable in a term.

```plaintext
fun term_t_bl_subs_for_i_var (Var v) = if i_var = v then term else Var v

term_t_bl_subs_for_i_var (μ t) = (μ (term_t_bl_subs_for_i_var t))

term_t_bl_subs_for_i_var (comp(v,f)) = if i_var = v then comp(v,f) else comp(v, term_bl_subs_for_i_var f)

term_t_bl_subs_for_i_var (V t) = V (term_t_bl_subs_for_i_var t)

term_t_bl_subs_for_i_var (Pair(f,s)) = Pair(term_t_bl_subs_for_i_var f, term_t_bl_subs_for_i_var s)

term_t_bl_subs_for_i_var (P t) = P (term_t_bl_subs_for_i_var t)

term_t_bl_subs_for_i_var (∪ t) = ∪ (term_t_bl_subs_for_i_var t)

term_t_bl_subs_for_i_var (∩ t) = ∩ (term_t_bl_subs_for_i_var t)

term_t_bl_subs_for_i_var (Nat i) = Nat i

term_t_bl_subs_for_i_var (Lit s) = Lit s

term_t_bl_subs_for_i_var Ø = Ø

and curried_t_subs term_i_var term’ = term_t_bl_subs_for_i_var term’
```
4.2.2. Into Formulae

Next we define substitution of a term for a free variable in a formula.

\[
\begin{align*}
\text{term_bl_subs_for_ivar_inn}(\text{Pvar } p) &= (\text{Pvar } p) & | \\
\text{term_bl_subs_for_ivar_inn}(\neg \text{form}) &= \neg (\text{term_bl_subs_for_ivar_inn} \text{form}) & | \\
\text{term_bl_subs_for_ivar_inn}(\text{form1 } \equiv \text{form2}) &= (\text{term_bl_subs_for_ivar_inn} \text{form1}) \equiv (\text{term_bl_subs_for_ivar_inn} \text{form2}) & | \\
\text{term_bl_subs_for_ivar_inn}(\text{form1 } \land \text{form2}) &= (\text{term_bl_subs_for_ivar_inn} \text{form1}) \land (\text{term_bl_subs_for_ivar_inn} \text{form2}) & | \\
\text{term_bl_subs_for_ivar_inn}(\text{form1 } \Rightarrow \text{form2}) &= (\text{term_bl_subs_for_ivar_inn} \text{form1}) \Rightarrow (\text{term_bl_subs_for_ivar_inn} \text{form2}) & | \\
\text{term_bl_subs_for_ivar_inn}(\forall (\text{ivar}', \text{form})) &= \text{if } \text{ivar} = \text{ivar}' \text{ then } \forall (\text{ivar}', \text{form}) \text{ else } \forall (\text{ivar}', \text{term_bl_subs_for_ivar_inn} \text{form}) & | \\
\text{term_bl_subs_for_ivar_inn}(\exists (\text{ivar}', \text{form})) &= \text{if } \text{ivar} = \text{ivar}' \text{ then } \exists (\text{ivar}', \text{form}) \text{ else } \exists (\text{ivar}', \text{term_bl_subs_for_ivar_inn} \text{form}) & | \\
\text{term_bl_subs_for_ivar_inn} (\text{t1 } == \text{t2}) &= (\text{term_t_bl_subs_for_ivar_inn} \text{t1}) == (\text{term_t_bl_subs_for_ivar_inn} \text{t2}) & | \\
\text{term_bl_subs_for_ivar_inn} (\text{t1 } \in \text{t2}) &= (\text{term_t_bl_subs_for_ivar_inn} \text{t1}) \in (\text{term_t_bl_subs_for_ivar_inn} \text{t2}) & | \\
\text{term_blsubs_for_ivar_inn} \text{T } \text{T} &= \\
\text{and } \text{curried_bl_subs } \text{abc } \text{b } \text{c } &= \text{a } \text{bl_subs_for } \text{b } \text{inn } \text{c};
\end{align*}
\]
4.3. Substitution of Formulae

4.3.1. Into Terms

The following function substitutes a formula (prop) for a propositional variable in a second formula. Renaming of bound variables occurs as necessary to avoid capture of free variables.

```haskell
fun prop t_psubs_for pvar inn (V arr v) = V arr v
prop t_psubs_for pvar inn (µ t) = (µ (prop t_psubs_for pvar inn t))
prop t_psubs_for pvar inn (comp(v,f)) = let
  val nv = primed v (freevars prop @
    (freevars f -- v))
  in
  (comp(nv,nf))
end
prop t_psubs_for pvar inn (V t) = V (prop t_psubs_for pvar inn t)
prop t_psubs_for pvar inn (Pair (f,s)) = Pair (prop t_psubs_for pvar inn f,
  prop t_psubs_for pvar inn s)
prop t_psubs_for pvar inn ( catchError (t)) = catchError (prop t_psubs_for pvar inn t)
prop t_psubs_for pvar inn (set_union (t)) = set_union (prop t_psubs_for pvar inn t)
prop t_psubs_for pvar inn (set_intersection (t)) = set_intersection (prop t_psubs_for pvar inn t)
prop t_psubs_for pvar inn (Nat i) = Nat i
prop t_psubs_for pvar inn (Lit s) = Lit s
prop t_psubs_for pvar inn ∅ = ∅
and curried_t_psubs prop pvar term’ = prop t_psubs_for pvar inn term’
```
4.3.2. Into Formulae

Next substitution of a formula (prop) for a propositional variable (p) in a formula is defined, renaming bound variables as necessary.

\[
\text{and prop psubs_for pvar inn (Pvar p)} = \text{prop} \\
\text{prop psubs_for pvar inn (¬ form)} = \neg (\text{prop psubs_for pvar inn form}) \\
\text{prop psubs_for pvar inn (form1 \ implies form2)} = (\text{prop psubs_for pvar inn form1}) \ implies (\text{prop psubs_for pvar inn form2}) \\
\text{prop psubs_for pvar inn (form1 \ and form2)} = (\text{prop psubs_for pvar inn form1}) \ and (\text{prop psubs_for pvar inn form2}) \\
\text{prop psubs_for pvar inn (form1 \ implies form2)} = (\text{prop psubs_for pvar inn form1}) \ implies (\text{prop psubs_for pvar inn form2}) \\
\text{prop psubs_for pvar inn (form1 \ and form2)} = (\text{prop psubs_for pvar inn form1}) \ and (\text{prop psubs_for pvar inn form2}) \\
\text{prop psubs_for pvar inn (form1 \ equivalent form2)} = (\text{prop psubs_for pvar inn form1}) \ equivalent (\text{prop psubs_for pvar inn form2}) \\
\text{prop psubs_for pvar inn (forall (ivar,form))} = \text{let val nv = primed ivar (freevars prop @ (freevars form -- ivar))} \\
\text{val nf = prop psubs_for pvar inn ((Var nv) subs_for ivar inn form)} \\
\text{in (forall (nv,nf))} \\
\text{end) \\
\text{prop psubs_for pvar inn (exists (ivar,form))} = \text{let val nv = primed ivar (freevars prop @ (freevars form -- ivar))} \\
\text{val nf = prop psubs_for pvar inn ((Var nv) subs_for ivar inn form)} \\
\text{in (exists (nv,nf))} \\
\text{end) \\
\text{prop psubs_for pvar inn (t1 == t2)} = (\text{prop t_psubs_for pvar inn t1}) == (\text{prop t_psubs_for pvar inn t2}) \\
\text{prop psubs_for pvar inn (t1 \in t2)} = (\text{prop t_psubs_for pvar inn t1}) \in (\text{prop t_psubs_for pvar inn t2}) \\
\text{prop psubs_for pvar inn T = T} \\
\text{and curried_psubs a b c = a psubs_for b inn c;}
\]
4.4. Blind Substitution of Formulae

4.4.1. Into Terms

For use in equivalence substitutions we require a variant of the procedure for substituting a formula for a propositional variable which allows capture of free (individual) variables.

```fsharp
fun prop_t_bl_psubs_for_pvar inn (Var v) = Var v
prop_t_bl_psubs_for_pvar inn (μ t) = (μ (prop_t_bl_psubs_for_pvar inn t))
prop_t_bl_psubs_for_pvar inn (comp(v,f)) = comp(v, prop bl_psubs_for_pvar inn f)
prop_t_bl_psubs_for_pvar inn (V t) = V (prop_t_bl_psubs_for_pvar inn t)
prop_t_bl_psubs_for_pvar inn (Pair (f,s)) = Pair ( prop_t_bl_psubs_for_pvar inn f, prop_t_bl_psubs_for_pvar inn s)
prop_t_bl_psubs_for_pvar inn (P t) = P (prop_t_bl_psubs_for_pvar inn t)
prop_t_bl_psubs_for_pvar inn (∪ t) = ∪ (prop_t_bl_psubs_for_pvar inn t)
prop_t_bl_psubs_for_pvar inn (∩ t) = ∩ (prop_t_bl_psubs_for_pvar inn t)
prop_t_bl_psubs_for_pvar inn (Nat i) = Nat i
prop_t_bl_psubs_for_pvar inn (Lit s) = Lit s
prop_t_bl_psubs_for_pvar inn ∅ = ∅
```

and         curried_t_bl_psubs prop pvar term’
        =prop t_bl_psubs_for_pvar inn term’
4.4.2. Into Formulae

\[
\begin{align*}
\text{prop bl_psubs_for pvar inn (Pvar p)} &= \text{prop} \\
\text{prop bl_psubs_for pvar inn (¬ form)} &= ¬\ (\text{prop bl_psubs_for pvar inn form)} \\
\text{prop bl_psubs_for pvar inn (form1 \lor form2)} &= (\text{prop bl_psubs_for pvar inn form1}) \lor (\text{prop bl_psubs_for pvar inn form2}) \\
\text{prop bl_psubs_for pvar inn (form1 \land form2)} &= (\text{prop bl_psubs_for pvar inn form1}) \land (\text{prop bl_psubs_for pvar inn form2}) \\
\text{prop bl_psubs_for pvar inn (∀ (ivar, form))} &= (∀ (ivar, \text{prop bl_psubs_for pvar inn form})) \\
\text{prop bl_psubs_for pvar inn (∃ (ivar, form))} &= (∃ (ivar, \text{prop bl_psubs_for pvar inn form})) \\
\text{prop bl_psubs_for pvar inn (t1 = t2)} &= (\text{prop t_bl_psubs_for pvar inn t1}) = (\text{prop t_bl_psubs_for pvar inn t2}) \\
\text{prop bl_psubs_for pvar inn (t1 ∈ t2)} &= (\text{prop t_bl_psubs_for pvar inn t1}) ∈ (\text{prop t_bl_psubs_for pvar inn t2}) \\
\text{prop bl_psubs_for pvar inn T = T} \\
\end{align*}
\]

and \( \text{curried_bl_psubs a b c = a bl_psubs_for b inn c} \);

5. DERIVED CONSTRUCTORS

5.1. Subset

\[
\begin{align*}
\text{fun a ⊆ b} &= \quad \text{let val x = freev [a,b] "x" in} \\
&\quad \quad \forall (x,(\text{Var x} \in a) \lor (\text{Var x} \in b)) \\
&\quad \text{end};
\end{align*}
\]

5.2. 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).

\[
\begin{align*}
\text{fun set a} &= \quad \text{let val x = (freev [a] "x") in} \\
&\quad \quad ∃(x, a ∈ (\text{Var x})) \\
&\quad \text{end};
\end{align*}
\]

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")); *)

5.3. Pairs Ordered Pairs and unit sets.

We define here a curried version of Pair, called pair, and give the usual definition of ordered pairs.

fun pair a b = Pair (a,b);

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

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

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

fun unit a = pair a a;

5.4. Power Set, Union, Intersection

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

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

(* fun ∩ a =
     let val [x,y] = map (freev [a]) ["x","y"]
     in comp (x,∀ (y,(Var y ∈ a) ∈ (Var x ∈ (Var y))))
     end; *)
5.5. Relations and (Partial) Functions

A relation is simply a set of ordered pairs.

fun is_rel a = let val \([w,x,y,z]\) = map (free v\[ a\]) \["w","x","y","z"\]
in  \(\forall (w, (\text{Var } w \in a) \Rightarrow \exists (x,\exists (y, ((\text{Var } w) == \text{opair (Var } x) (\text{Var } y))))))\)
end;

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

fun dom f = let val \([x,y]\) = map (free v\[ f\]) \["x","y"\]
in  \(\text{comp (x,}\exists (y,\text{opair (Var } x) (\text{Var } y) \in f))\)
end;

fun ran f = let val \([x,y]\) = map (free v\[ f\]) \["x","y"\]
in  \(\text{comp (x,}\exists (y,\text{opair (Var } y) (\text{Var } x) \in f))\)
end;

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

fun is_sv a = let val \([x,y,z]\) = map (free v\[ a\]) \["x","y","z"\]
in  is_rel a \(\land \forall (x,\forall (y,\forall (z,\text{opair (Var } x) (\text{Var } y) \in a) \land \text{opair (Var } x) (\text{Var } z) \in a) \Rightarrow ((\text{Var } y) == (\text{Var } z))))\)
end;

The following function is the domain co-restriction of a relation. The domain restriction of \(r\) to \(d\) is the class of ordered pairs in \(r\) whose left member is not in \(d\).

fun dom_corestr r d = let val x = free v\[ r, d\] "x"
in  \(\text{comp(x, (\text{Var } x) \in r \land (\text{left (Var } x) \in d))}\)
end;

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

fun rel_over r1 r2 = \(\cup (\text{pair (dom_corestr r1 (dom r2)) r2})\);

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

fun rel_update r1 a v = rel_over r1 (opair a v);
6. 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.

6.1. Inference Rules

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

structure THM = struct
local open FTM in
abstype theorem = of formula
with

6.1.1. Modus Ponens

Modus Ponens takes two theorems of the form \( A, A \implies B \) 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)(B,C) = if A=B then C else A | MP x y = x;

6.1.2. Generalisation

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

fun UI (A) x = (\forall(x,A));

6.1.3. Propositional Substitution

Any formula may be substituted in place of a propositional variable in any theorem. Free variables in the formula being substituted will cause renaming of bound variables if the propositional variable occurs in the scope of such a binding.

fun PS f pvar (t) = (f psubs_for pvar inn t);

6.1.4. Equivalence Substitution

Any formula (rhf) may be substituted in place of a formula provably equivalent to it (lhf), in any theorem (sf) yielding a theorem. This is possible even if variables free in the formulae rhf,lhf are bound in the premiss sf or the conclusion.
fun EQVS f pv (sf) ((op ⇔ (lhf, rhf))) =  
    if lhf bl_psubs_for pv inn f = sf  
    then (rhf bl_psubs_for pv inn f)  
    else T;

6.1.5. Equality Substitution
Any term may be substituted in place of a term provably equal to it.

fun EQLS f v (sf) ((op == (lht, rht))) =  
    if lht bl_subs_for v inn f = sf  
    then (rht bl_subs_for v inn f)  
    else T;

6.2. Propositional Axioms
All pure tautologies are axioms.
Any pure tautology is an axiom. A pure tautology is a tautology in which all atomic formulae are either propositional variables or T.
Function 'tautc' checks a formula to determine whether it is a pure tautology.

fun tautcxT = true |  
    tautc [] f = false |  
    tautc (h::t) f =  
        (tautc t (simpl h T f))  
        andalso (tautc t (simpl h F f));

The inference rule TAU TC will return a theorem given a formula which is a pure tautology.

fun TAUT f =  
    if tautc (pack_set ((propvars f)@[""])) f = true  
    then f  
    else T;

6.3. 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)));  
fun Q4 A x = (∃(x,A) ⇔ ¬(∀(x,¬A)));

DBC/FMU/RBJ/116 Issue 0.6 UST mk 2
6.4. Equality and Membership

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

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

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

6.5. Term Constructor Axioms

6.5.1. Pair

6.5.2. Union

6.5.3. Intersection

6.5.4. Natural Numbers

```ml
fun suc n = ∪ (pair n (unit n));
val NAT0 = ((Nat 0) == Ø);
fun NAT n = if n<0
    then (Nat (0-n) == Nat n)
    else (Nat (n+1) == ∪ (pair (Nat n) (unit (Nat n))));
```

6.5.5. Strings

```ml
fun LIT "" = ((Lit ")") == Ø) |
    LIT s = case explode s of
    [] => ((Lit ")") == Ø) |
    (h::t) => ((Lit (implode (h::t)))
        == opair (Nat (ord h)) (Lit (implode t)));
```

6.5.6. The empty set

```ml
val ES_AX = (Ø == comp(x,F));
```
6.6. 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 fol-
lowing axiom states, it yields a class which contains just those sets which satisfy the predicate
(omitting any classes which satisfy the predicate). Proper classes are not members of anything
(that is the definition of a proper class), and the argument about whether the Russell class is a
member of itself proves that the class is proper rather than giving rise to a contradiction.

\[
\text{fun COM } x \ p = (\forall x, \text{set } \text{Var } x \land p \iff \text{Var } x \in \text{comp}(x, p));
\]

6.7. The Empty Set

The empty class is a set.

\[
\text{fun } S \emptyset x = \text{set } \emptyset;
\]

6.8. 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
pair and 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 state-
ment 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.

\[
\begin{align*}
\text{fun } \text{Umem } u & = (\text{set } u \in (V u) \land \text{set } (V u)); \\
\text{fun } \text{Utrans } u x & = (x \in (V u) \land x \subseteq (V u)); \\
\text{fun } \text{Upair } u x y & = (x \in (V u) \land (x \in (V u)) \\
& \text{Pair } (x, y) \in V u); \\
\text{fun } \text{Upower } u a & = (a \in (V u) \land (\text{P } a \in (V u))); \\
\text{fun } \text{Union } u a & = (a \in (V u) \land (\cup a \in (V u))); \\
\text{fun } \text{Urep } u f = (\text{is } \text{sv } f \land (\text{dom } f) \in (V u) \land (\text{ran } f) \subseteq (V u) \land (\text{ran } f) \in (V u));
\end{align*}
\]

6.9. 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 $\mu \emptyset$, which is by this axiom a proper class and hence may not be in the range of any function).

```sml
fun CH t = ((set (\mu t) \iff (\neg (t == \emptyset)))
    \land ((\neg (t == \emptyset)) \iff (\mu t) \in t));
```

### 6.10. Foundation

The axiom of foundation is needed to permit inductive definitions.

```sml
fun FO t = let val y = primed "y" (term_freevars t)
    in (t==\emptyset
        \exists(y,\Var y \in t \land (\cap (pair t (\Var y)) == \emptyset)))
    end;
```

### 6.11. Extracting the Formula from a Theorem

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

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

### 6.12. End of Abstract Data Type

```sml
end; (* of local clause *)
end; (* of abstract data type *)
end; (* of structure THM *)
end; (* of structure UST *)
```

The types inferred by the SML compiler were: