1. INTRODUCTION

This document describes the semantics of the types of VDM. It does so using the language SML (standard ML) to define first the abstract syntax of first order set theory (ZFC), and then a useful collection of terms of ZFC and operators over these terms. These terms are taken to denote their meaning in the intended interpretation of ZFC.

We should then be in a position to define the abstract syntax of type expressions, and a mapping from this language into terms of ZFC.

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

2. ABSTRACT SYNTAX

```
infix 5 \in
infix 6 \%
infix 7 :!
infix 6 !: == \epsilon;
type var = string;
```
datatype
typed var = op ! of (var * term)
and separation = op : of (typed var * formula)
and formula = T of term
¬ of formula
op \( \not\in \) of (formula * formula)
\( \exists \) of (var * formula)
op == of (term * term)
op \( \in \) of (term * term)
and term =
V of var
\( \emptyset \)
Nat
op \% of (term * term)
\( \mathbb{P} \) of term
\( \mu \) of term
\( \cup \) of term
\$ of separation;

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.

```sml
infix 8 \in;
fun x \in [] = false |
   x \in (y::z) = x=y orelse x \in z;

infix 8 \subseteq;
fun [] \subseteq y = true |
   (h::t) \subseteq y = h \in y andalso t \subseteq y;

fun m \cup [] = [] |
   m \cup (a::b) = a @ (m \cup b);

infix 8 --;
fun [] -- a = [] |
   (h::t) -- a = if h=a then t -- a else h::(t -- a);

infix 8 \setminus;
fun a \setminus [] = a |
   a \setminus (h::t) = (a -- h) \setminus t;
```
### 3.2. Free Variables

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

\[
\begin{align*}
\text{fun } & \text{term_freevars } (V \ v) = [v] \\
& \text{term_freevars } \emptyset = [] \\
& \text{term_freevars } (a \% b) = \text{term_freevars } a @ (\text{term_freevars } b) \\
& \text{term_freevars } (\mathcal{P} \ t) = \text{term_freevars } t \\
& \text{term_freevars } (\mu \ t) = \text{term_freevars } t \\
& \text{term_freevars } (\cup \ t) = \text{term_freevars } t \\
& \text{term_freevars } ((v : ! f : f)) = (\text{term_freevars } t) @ (\text{freevars } f -- v) \\
\end{align*}
\]

\[
\begin{align*}
\text{and } & \text{term_list_freevars } \text{term_list } = m \cup (\text{map } \text{term_freevars } \text{term_list}) \\
& \text{freevars } (T \ t) = \text{term_freevars } t \\
& \text{freevars } (\neg f) = \text{freevars } f \\
& \text{freevars } (f_1 \equiv f_2) = \text{freevars } f_1 @ (\text{freevars } f_2) \\
& \text{freevars } (\exists (v,f)) = \text{freevars } f -- v \\
& \text{freevars } (t_1 == t_2) = \text{term_freevars } t_1 @ (\text{term_freevars } t_2) \\
& \text{freevars } (t_1 \in t_2) = \text{term_freevars } t_1 @ (\text{term_freevars } t_2);
\end{align*}
\]

### 3.3. Substitutions

First define substitutions into terms.

\[
\begin{align*}
\text{fun } & \text{primed } \text{nv varl } = \text{if } \text{nv } \varepsilon \text{ varl} \\
& \text{then } \text{primed } (\text{nv }^{"\ast"}) \text{ varl} \\
& \text{else } \text{nv};
\end{align*}
\]

\[
\text{infix } 9 \ \text{inn};
\]

\[
\text{datatype } ('a)\text{inc } = \text{op inn of } ('a * 'a);
\]
infix 8 t_subs_for;
fun term t_subs_for i var inn (V v) =
  if i var = v then term
  else V v
   |
term t_subs_for i var inn ∅ = ∅ |
term t_subs_for i var inn Nat = Nat |
term t_subs_for i var inn (t1 % t2) =
  term t_subs_for i var inn t1 % (term t_subs_for i var inn t2) |
term t_subs_for i var inn (P t) =
  (P (term t_subs_for i var inn t)) |
term t_subs_for i var inn (µ t) =
  (µ (term t_subs_for i var inn t)) |
term t_subs_for i var inn (∪ t) =
  (∪ (term t_subs_for i var inn t)) |
term t_subs_for i var inn ($v:!t!:f) =
  if i var = v then $v!:(term t_subs_for i var inn t)!f)
  else let val nv = primed v (term_freevars term @
    (freevars f -- v))
    and nt = term t_subs_for i var inn t
    and nf = term_subs_for v inn
    ((V v) subs_for v inn f)
    in $nv!:nt!:nf
  end
end
and curried_t_subs term i var term’ =
  term t_subs_for i var inn term’

Next substitutions into formulae.
infix 8 subs_for;
and term_subs_for_i var nn (T t) = T (term_subs_for_i var nn t)  
term_subs_for_i var nn (¬ form) = ¬ (term_subs_for_i var nn form)  
term_subs_for_i var nn (form1 ≜ form2) = (term_subs_for_i var nn form1) ≜ (term_subs_for_i var nn form2)  
term_subs_for_i var nn (∃ (i var’,form)) = (∃ (i var’,term_subs_for_i var nn form))  
term_subs_for_i var nn (t1 == t2) = (term t_subs_for_i var nn t1) == (term t_subs_for_i var nn t2)  
term_subs_for_i var nn (t1 ∈ t2) = (term t_subs_for_i var nn t1) ∈ (term t_subs_for_i var nn t2)  
and curried_subs a b c = a subs_for b inn c;

4. DERIVED CONSTRUCTORS

let fun unit s = s % s;;

5. BASIC TYPES

For the time being I will leave open the question of what are the basic types. To enable the subsequent definitions to be type checked I will define Bty to be a set containing only Nat, the natural numbers.

let val bty = unit Nat;;

6. OPERATORS

The operators are defined over all sets, and may not therefore be defined as sets. We define them as formulae expressing relationships between sets.