# resources

Leo Freitas

June 8, 2022

# Contents

**theory** *VDMEisbach*
**imports** *Complex-Main*
**begin**

**named-theorems**
*VDM-basic-defs* **and**

*VDM-num-defs* **and**
*VDM-num-fcns* **and**
*VDM-num-spec-pre* **and**
*VDM-num-spec-post* **and**
*VDM-num-spec* **and**
*VDM-num* **and**

*VDM-set-defs* **and**
*VDM-set-fcns* **and**
*VDM-set-spec-pre* **and**
*VDM-set-spec-post* **and**
*VDM-set-spec* **and**
*VDM-set* **and**

*VDM-seq-defs* **and**
*VDM-seq-fcns-1* **and**
*VDM-seq-fcns-2* **and**
*VDM-seq-fcns-3* **and**
*VDM-seq-fcns* **and**
*VDM-seq-spec-pre* **and**
*VDM-seq-spec-post-1* **and**
*VDM-seq-spec-post-2* **and**
*VDM-seq-spec-post-3* **and**
*VDM-seq-spec-post* **and**
*VDM-seq-spec* **and**
*VDM-seq* **and**

*VDM-map-defs* **and**

*VDM-map-fcns-simps* **and**
*VDM-map-fcns-1-simps* **and**
*VDM-map-fcns-2-simps* **and**

*VDM-map-fcns* **and**
*VDM-map-fcns-1* **and**
*VDM-map-fcns-2* **and**
*VDM-map-fcns-3* **and**

*VDM-map-fcns-4* **and**

*VDM-map-comp* **and**
*VDM-map-comp-1* **and**
*VDM-map-comp-2* **and**
*VDM-map-comp-3* **and**

*VDM-map* **and**

*VDM-num-crc* **and**
*VDM-num-crc-1* **and**
*VDM-num-crc-2* **and**
*VDM-num-crc-3* **and**

*VDM-stms-defs* **and**
*VDM-stms* **and**

*VDM-spec* **and**
*VDM-all*

**end**

**theory** *VDMToolkit*
**imports**
— Include real fields, list and option types ordering
*Complex-Main*
*VDMEisbach*
*HOL−Library.List-Lexorder*
*HOL−Library.Option-ord*
*HOL−Library.LaTeXsugar*
*HOL−Library.While-Combinator*
**begin**

# 1 Basic types

**type-notation** *bool* ($\mathbb{B}$)
**type-notation** *nat* ($\mathbb{N}$)
**type-notation** *int* ($\mathbb{Z}$)
**type-notation** *rat* ($\mathbb{Q}$)
**type-notation** *real* ($\mathbb{R}$)

VDM numeric expressions have a series of implicit type widening rules. For example, $4 - x$ could lead to an integer $- y$ result, despite all parameters involved being $\mathbb{N}$, whereas in HOL, the result is always a $\mathbb{N}$ ultimately equal to$0{::}'a$.

Therefore, we take the view of the widest (compatible) type to use in the translation, where type widening to $\mathbb{Q}$ or $\mathbb{R}$ is dealt with through Isabelle's type coercions.

**type-synonym** $\mathit{VDMNat} = \mathbb{Z}$
**type-synonym** $\mathit{VDMNat1} = \mathbb{Z}$
**type-synonym** $\mathit{VDMInt} = \mathbb{Z}$
**type-synonym** $\mathit{VDMRat} = \mathbb{Q}$
**type-synonym** $\mathit{VDMReal} = \mathbb{R}$
**type-synonym** $\mathit{VDMChar} = \mathit{char}$

Moreover, VDM type invariant checks have to be made explicit in VDM. That is possible either through subtyping, which will require substantial proof-engineering machinery; or through explicit type invariant predicates. We choose the later for all VDM types.

**definition**
$\quad \mathit{inv\text{-}VDMNat} :: \mathbb{Z} \Rightarrow \mathbb{B}$
**where**
$\qquad \mathit{inv\text{-}VDMNat}\ n \equiv n \geq \mathit{0}$

**definition**
$\quad \mathit{inv\text{-}VDMNat1} :: \mathbb{Z} \Rightarrow \mathbb{B}$
**where**
$\qquad \mathit{inv\text{-}VDMNat1}\ n \equiv n > \mathit{0}$

Bottom invariant check is that value is not undefined.

**definition**
$\quad \mathit{inv\text{-}True} :: {'a} \Rightarrow \mathbb{B}$
$\quad$ **where**
$\quad [\mathit{intro}!]\text{: } \mathit{inv\text{-}True} \equiv \lambda\ x\ .\ \mathit{True}$

**definition**
$\quad \mathit{inv\text{-}bool} :: \mathbb{B} \Rightarrow \mathbb{B}$
**where**
$\qquad \mathit{inv\text{-}bool}\ i \equiv \mathit{inv\text{-}True}\ i$

**definition**
$\quad \mathit{inv\text{-}VDMChar} :: \mathit{VDMChar} \Rightarrow \mathbb{B}$
**where**
$\qquad \mathit{inv\text{-}VDMChar}\ c \equiv \mathit{inv\text{-}True}\ c$

**definition**
$\quad \mathit{inv\text{-}VDMInt} :: \mathbb{Z} \Rightarrow \mathbb{B}$
**where**
$\qquad \mathit{inv\text{-}VDMInt}\ i \equiv \mathit{inv\text{-}True}\ i$

**definition**
$\quad \mathit{inv\text{-}VDMReal} :: \mathbb{R} \Rightarrow \mathbb{B}$
**where**
$\qquad \mathit{inv\text{-}VDMReal}\ r \equiv \mathit{inv\text{-}True}\ r$

**definition**

$inv\text{-}VDMRat :: \mathbb{Q} \Rightarrow \mathbb{B}$
**where**
$inv\text{-}VDMRat\ r \equiv inv\text{-}True\ r$

**lemma** $l\text{-}inv\text{-}True\text{-}True[simp]$: $inv\text{-}True\ r$
**by** $(simp\ add: inv\text{-}True\text{-}def)$

In general, VDM narrow expressions are tricky, given they can downcast types according to the user-specified type of interest. In particular, at least for $\mathbb{R}$ and $\mathbb{Q}$ (*floor-ceiling* type class), type narrowing to *VDMInt* is fine

**definition**
$vdm\text{-}narrow\text{-}real :: ('a::floor\text{-}ceiling) \Rightarrow VDMInt$
**where**
$vdm\text{-}narrow\text{-}real\ r \equiv \lfloor r \rfloor$

**definition**
$vdm\text{-}div :: VDMInt \Rightarrow VDMInt \Rightarrow VDMInt$ (**infixl** $vdmdiv\ 70$)
**where**
$[intro!]$ :
$x\ vdmdiv\ y \equiv$
$(if\ ((x\ /\ y) < 0)\ then$
$-\lfloor |-x\ /\ y| \rfloor$
$else$
$\lfloor |x\ /\ y| \rfloor)$

**definition**
$pre\text{-}vdm\text{-}div :: VDMInt \Rightarrow VDMInt \Rightarrow \mathbb{B}$
**where**
$pre\text{-}vdm\text{-}div\ x\ y \equiv y \neq 0$

**definition**
$post\text{-}vdm\text{-}div :: VDMInt \Rightarrow VDMInt \Rightarrow VDMInt \Rightarrow \mathbb{B}$
**where**
$post\text{-}vdm\text{-}div\ x\ y\ RESULT \equiv$
$(x \geq 0 \land y \geq 0 \longrightarrow RESULT \geq 0) \land$
$(x < 0 \land y < 0 \longrightarrow RESULT \geq 0) \land$
$(x < 0 \land 0 < y \longrightarrow RESULT \leq 0) \land$
$(0 < x \land y < 0 \longrightarrow RESULT \leq 0)$

VDM has div and mod but also rem for remainder. This is treated differently depending on whether the values involved have different sign. For now, we add these equivalences below, but might have to pay price in proof later. To illustrate this difference consider the result of $-\ 7\ div\ 3 = -\ 3$ versus $-\ 7\ vdmdiv\ 3 = -\ 2$

**definition**
$vdm\text{-}mod :: VDMInt \Rightarrow VDMInt \Rightarrow VDMInt$ (**infixl** $vdmmod\ 70$)

**where**
  [*intro*!] :
  $x\ vdmmod\ y \equiv x - y * \lfloor x\ /\ y \rfloor$

**definition**
  $pre\text{-}vdm\text{-}mod :: VDMInt \Rightarrow VDMInt \Rightarrow \mathbb{B}$
  **where**
  $pre\text{-}vdm\text{-}mod\ x\ y \equiv y \neq 0$

**definition**
  $post\text{-}vdm\text{-}mod :: VDMInt \Rightarrow VDMInt \Rightarrow VDMInt \Rightarrow \mathbb{B}$
  **where**
  $post\text{-}vdm\text{-}mod\ x\ y\ RESULT \equiv$
    $(y \geq 0 \longrightarrow RESULT \geq 0) \wedge$
    $(y < 0 \longrightarrow RESULT \leq 0)$

**definition**
  $vdm\text{-}rem :: VDMInt \Rightarrow VDMInt \Rightarrow VDMInt$ (**infixl** *vdmrem 70*)
**where**
  [*intro*!] :
  $x\ vdmrem\ y \equiv x - y * (x\ vdmdiv\ y)$

**definition**
  $pre\text{-}vdm\text{-}rem :: VDMInt \Rightarrow VDMInt \Rightarrow \mathbb{B}$
  **where**
  $pre\text{-}vdm\text{-}rem\ x\ y \equiv y \neq 0$

**definition**
  $post\text{-}vdm\text{-}rem :: VDMInt \Rightarrow VDMInt \Rightarrow VDMInt \Rightarrow \mathbb{B}$
  **where**
  $post\text{-}vdm\text{-}rem\ x\ y\ RESULT \equiv$
    $(x \geq 0 \longrightarrow RESULT \geq 0) \wedge$
    $(x < 0 \longrightarrow RESULT \leq 0)$

VDM has the power (`**`) operator for numbers, which is (*powr*) in Issable. Like in VDM, it accepts non-integer exponents. Isabelle have $x^y$ for exponent $y$ of type $\mathbb{N}$, and $x\ powr\ y$ for exponent $y$ that is a subset of the $\mathbb{R}$ (i.e. real normed algebra natural logarithms; or natural logarithm exponentiation). We take the latter for translation.

**definition**
  $vdm\text{-}pow :: 'a{::}ln \Rightarrow 'a{::}ln \Rightarrow 'a{::}ln$ (**infixl** *vdmpow 80*)
  **where**
  [*intro*!]: $x\ vdmpow\ y \equiv x\ powr\ y$

**definition**
  $pre\text{-}vdm\text{-}pow :: 'a{::}ln \Rightarrow 'a{::}ln \Rightarrow \mathbb{B}$
  **where**
  $pre\text{-}vdm\text{-}pow\ x\ y \equiv True$

**definition**
$post\text{-}vdm\text{-}pow :: 'a::ln \Rightarrow 'a::ln \Rightarrow 'a::ln \Rightarrow \mathbb{B}$
**where**
$post\text{-}vdm\text{-}pow\ x\ y\ RESULT \equiv True$

For VDM floor and abs, we use Isabelle's. Note that in VDM abs of $\mathbb{Z}$ will return *VDMNat*, as the underlying type invariant might require further filtering on the function's results.

**find-theorems** - (-::'*a list list*) *name*:*concat*
**definition**
$vdm\text{-}floor :: VDMReal \Rightarrow VDMNat$
**where**
[*intro*!]: $vdm\text{-}floor\ x \equiv \lfloor x \rfloor$

The postcondition for flooring, takes the axiom defined in the archimedian field type class

**definition**
$post\text{-}vdm\text{-}floor :: VDMReal \Rightarrow VDMNat \Rightarrow \mathbb{B}$
**where**
$post\text{-}vdm\text{-}floor\ x\ RESULT \equiv$
$of\text{-}int\ RESULT \leq x \wedge x < of\text{-}int\ (RESULT + 1)$

**definition**
$vdm\text{-}abs :: ('a::\{zero,abs,ord\}) \Rightarrow ('a::\{zero,abs,ord\})$
**where**
[*intro*!]: $vdm\text{-}abs\ x \equiv |x|$

Absolute postcondition does not use *inv-VDMNat* because the result could also be of type $\mathbb{R}$.

**definition**
$post\text{-}vdm\text{-}abs :: ('a::\{zero,abs,ord\}) \Rightarrow ('a::\{zero,abs,ord\}) \Rightarrow \mathbb{B}$
**where**
$post\text{-}vdm\text{-}abs\ x\ RESULT \equiv RESULT \geq 0$

For equally signed operands of VDM's div/mod, we can get back to Isabelle's version of the operators, which will give access to various lemmas useful in proofs. So, if possible, automatically jump to the Isabelle versions.

**lemma** *vdmdiv-div-ge0*[*simp*] :
$0 \leq x \Longrightarrow 0 \leq y \Longrightarrow x\ vdmdiv\ y = x\ div\ y$
**unfolding** *vdm-div-def*
**apply** (*induct y*) **apply** *simp-all*
**by** (*metis divide-less-0-iff floor-divide-of-int-eq floor-less-zero floor-of-int floor-of-nat le-less-trans less-irrefl of-int-of-nat-eq of-nat-less-0-iff*)

**lemma** *vdmdiv-div-le0*[*simp*] :
$x \leq 0 \Longrightarrow y \leq 0 \Longrightarrow x\ vdmdiv\ y = x\ div\ y$
**unfolding** *vdm-div-def*
**apply** (*induct y*) **apply** *simp-all*

```
  apply safe
   apply (simp add: divide-less-0-iff)
  by (metis (no-types, hide-lams) floor-divide-of-int-eq minus-add-distrib minus-divide-right
of-int-1 of-int-add of-int-minus of-int-of-nat-eq uminus-add-conv-diff)

lemma vdmmod-mod[simp] :
  x vdmmod y = x mod y
  unfolding vdm-mod-def
  apply (induct y) apply simp-all
   apply (metis floor-divide-of-int-eq minus-mult-div-eq-mod of-int-of-nat-eq)
  by (smt (verit, ccfv-threshold) floor-divide-of-int-eq minus-div-mult-eq-mod mult.commute
of-int-diff of-int-eq-1-iff of-int-minus of-int-of-nat-eq)

lemma l-vdm-div-fsb: pre-vdm-div x y ⟹ post-vdm-div x y (x vdmdiv y)
  unfolding pre-vdm-div-def post-vdm-div-def
  apply (safe)
  using div-int-pos-iff vdmdiv-div-ge0 apply presburger
  using vdm-div-def apply (smt (verit) divide-neg-neg floor-less-iff of-int-0-less-iff
of-int-minus)
  using vdm-div-def using divide-less-0-iff apply auto[1]
  using vdm-div-def
  by auto

lemma l-vdm-mod-fsb: pre-vdm-mod x y ⟹ post-vdm-mod x y (x vdmmod y)
  unfolding pre-vdm-mod-def post-vdm-mod-def
  apply safe
  by (simp add: vdm-mod-def)+

lemma l-vdm-rem-fsb: pre-vdm-rem x y ⟹ post-vdm-rem x y (x vdmrem y)
  unfolding pre-vdm-rem-def post-vdm-rem-def vdm-rem-def
  apply safe
  apply (cases y ≥ 0)
    apply simp
     apply (metis Euclidean-Division.pos-mod-sign add.commute add.left-neutral
add-mono-thms-linordered-semiring(3) div-mult-mod-eq le-less mult.commute)
   defer
   apply (cases y ≤ 0)
    apply simp
    apply (metis div-mod-decomp-int group-cancel.rule0 le-add-same-cancel1 le-less
mult.commute neg-mod-sign not-le)
  unfolding vdm-div-def
   apply (simp-all, safe)
     apply (smt (verit, ccfv-SIG) divide-minus-left floor-divide-lower floor-less-iff
floor-uminus-of-int mult.commute of-int-mult)
    apply (simp add: divide-neg-pos)
  apply (smt (verit) ceiling-def ceiling-divide-eq-div minus-mod-eq-mult-div neg-mod-sign)
  using divide-pos-neg by force
```

### 1.1 VDM tokens

VDM tokens are like a record with a parametric type (i.e. you can have anything inside a `mk_token(x)` expression, akin to a VDM record `Token :: token : ?`, where `?` refers to `vdmj` wildcard type. Isabelle does not allow parametric records, hence we use datatypes instead.

This will impose the restriction on token variables during translation: they will always have to be of the same inner type; whereas for token constants, then any type is acceptable.

**datatype** $'a$ *VDMToken* $=$ *Token* $'a$

**definition**
*inv-VDMToken* $::$ $'a$ *VDMToken* $\Rightarrow \mathbb{B}$
**where**
*inv-VDMToken* $t \equiv$ *inv-True* $t$

**definition**
*inv-VDMToken*$'$ $::$ $('a \Rightarrow \mathbb{B}) \Rightarrow$ $'a$ *VDMToken* $\Rightarrow \mathbb{B}$
**where**
*inv-VDMToken*$'$ *inv-T* $t \equiv$ *case* $t$ *of* *Token* $a \Rightarrow$ *inv-T* $a$

Isabelle lemmas definitions are issues for all the inner calls and related definitions used within given definitions. This allows for a laywered unfolding and simplification of VDM terms during proofs.

**lemmas** *inv-VDMToken$'$-defs* $=$ *inv-VDMToken$'$-def inv-True-def*

**lemma** *l-inv-VDMTokenI*[*simp*]: *inv-T* $a \Longrightarrow t = ($*Token* $a) \Longrightarrow$ *inv-VDMToken*$'$ *inv-T* $t$
**by** (*simp add*: *inv-VDMToken$'$-def*)

# 2 Sets

All VDM structured types (e.g. sets, sequences, maps, etc.) must check the type invariant of its constituent parts, beyond any user-defined invariant.

Moreover, all VDM sets are finite. Therefore, we define VDM set invariant checks as combination of finiteness checks with invariant checks of its elements type.

**type-synonym** $'a$ *VDMSet* $=$ $'a$ *set*
**type-synonym** $'a$ *VDMSet1* $=$ $'a$ *set*

**definition**
*inv-VDMSet* $::$ $'a$ *VDMSet* $\Rightarrow \mathbb{B}$
**where**
[*intro*!]: *inv-VDMSet* $s \equiv$ *finite* $s$

**definition**

$inv\text{-}VDMSet1 :: 'a\ VDMSet1 \Rightarrow \mathbb{B}$
**where**
$[intro!]:\ inv\text{-}VDMSet1\ s \equiv inv\text{-}VDMSet\ s \wedge s \neq \{\}$

**definition**
$inv\text{-}SetElems :: ('a \Rightarrow \mathbb{B}) \Rightarrow 'a\ VDMSet \Rightarrow \mathbb{B}$
**where**
$inv\text{-}SetElems\ einv\ s \equiv \forall\ e \in s\ .\ einv\ e$

Added wrapped version of the definition so that we can translate complex structured types (e.g. `seq of seq of T`, etc.). Parameter order matter for partial instantiation (e.g. $inv\text{-}VDMSet'\ (inv\text{-}VDMSet'\ inv\text{-}VDMNat)\ s$).

**definition**
$inv\text{-}VDMSet' :: ('a \Rightarrow \mathbb{B}) \Rightarrow 'a\ VDMSet \Rightarrow \mathbb{B}$
**where**
$[intro!]:\ inv\text{-}VDMSet'\ einv\ s \equiv inv\text{-}VDMSet\ s \wedge inv\text{-}SetElems\ einv\ s$

**definition**
$inv\text{-}VDMSet1' :: ('a \Rightarrow \mathbb{B}) \Rightarrow 'a\ VDMSet1 \Rightarrow \mathbb{B}$
**where**
$[intro!]:\ inv\text{-}VDMSet1'\ einv\ s \equiv inv\text{-}VDMSet1\ s \wedge inv\text{-}SetElems\ einv\ s$

**definition**
$vdm\text{-}card :: 'a\ VDMSet \Rightarrow VDMNat$
**where**
$vdm\text{-}card\ s \equiv (if\ inv\text{-}VDMSet\ s\ then\ int\ (card\ s)\ else\ undefined)$

**definition**
$pre\text{-}vdm\text{-}card :: 'a\ VDMSet \Rightarrow \mathbb{B}$
**where**
$[intro!]:\ pre\text{-}vdm\text{-}card\ s \equiv inv\text{-}VDMSet\ s$

**definition**
$post\text{-}vdm\text{-}card :: 'a\ VDMSet \Rightarrow VDMNat \Rightarrow \mathbb{B}$
**where**
$[intro!]:\ post\text{-}vdm\text{-}card\ s\ RESULT \equiv pre\text{-}vdm\text{-}card\ s \longrightarrow inv\text{-}VDMNat\ RESULT$

**lemmas** *inv-VDMSet-defs* = *inv-VDMSet-def*
**lemmas** *inv-VDMSet1-defs* = *inv-VDMSet1-def inv-VDMSet-def*
**lemmas** *inv-VDMSet′-defs* = *inv-VDMSet′-def inv-VDMSet-def inv-SetElems-def*
**lemmas** *inv-VDMSet1′-defs* = *inv-VDMSet1′-def inv-VDMSet1-defs inv-SetElems-def*
**lemmas** *vdm-card-defs* = *vdm-card-def inv-VDMSet-defs*

**lemma** *l-invVDMSet-finite-f*: $inv\text{-}VDMSet\ s \Longrightarrow finite\ s$
**using** *inv-VDMSet-def* **by** *auto*

**lemma** *l-inv-SetElems-Cons*[*simp*]: $(inv\text{-}SetElems\ f\ (insert\ a\ s)) = (f\ a \wedge (inv\text{-}SetElems\ f\ s))$
**unfolding** *inv-SetElems-def*

**by** *auto*

**lemma** *l-inv-SetElems-Un*[*simp*]: (*inv-SetElems f* ($S \cup T$)) = (*inv-SetElems f S* $\wedge$ *inv-SetElems f T*)
**unfolding** *inv-SetElems-def*
**by** *auto*

**lemma** *l-inv-SetElems-Int*[*simp*]: (*inv-SetElems f* ($S \cap T$)) = (*inv-SetElems f* ($S \cap T$))
**unfolding** *inv-SetElems-def*
**by** *auto*

**lemma** *l-inv-SetElems-empty*[*simp*]: *inv-SetElems f* {}
**unfolding** *inv-SetElems-def* **by** *simp*

**lemma** *l-invSetElems-inv-True-True*[*simp*]: *undefined* $\notin$ *r* $\Longrightarrow$ *inv-SetElems inv-True r*
**by** (*metis inv-SetElems-def l-inv-True-True*)

**lemma** *l-vdm-card-finite*[*simp*]: *finite s* $\Longrightarrow$ *vdm-card s* = *int* (*card s*)
**unfolding** *vdm-card-defs* **by** *simp*

**lemma** *l-vdm-card-range*[*simp*]: $x \leq y \Longrightarrow$ *vdm-card* $\{x\ ..\ y\} = y - x + 1$
**unfolding** *vdm-card-defs* **by** *simp*

**lemma** *l-vdm-card-positive*[*simp*]:
*finite s* $\Longrightarrow$ $0 \leq$ *vdm-card s*
**by** *simp*

**lemma** *l-vdm-card-VDMNat*[*simp*]:
*finite s* $\Longrightarrow$ *inv-VDMNat* (*vdm-card s*)
**by** (*simp add*: *inv-VDMSet-def inv-VDMNat-def*)

**lemma** *l-vdm-card-non-negative*[*simp*]:
*finite s* $\Longrightarrow s \neq \{\} \Longrightarrow 0 <$ *vdm-card s*
**by** (*simp add*: *card-gt-0-iff*)

**lemma** *l-vdm-card-isa-card*[*simp*]:
*finite s* $\Longrightarrow$ *card s* $\leq i \Longrightarrow$ *vdm-card s* $\leq i$
**by** *simp*

**lemma** *l-isa-card-inter-bound*:
*finite T* $\Longrightarrow$ *card T* $\leq i \Longrightarrow$ *card* ($S \cap T$) $\leq i$
**thm** *card-mono inf-le2 le-trans card-seteq Int-commute nat-le-linear*
**by** (*meson card-mono inf-le2 le-trans*)

**lemma** *l-vdm-card-inter-bound*:
*finite T* $\Longrightarrow$ *vdm-card T* $\leq i \Longrightarrow$ *vdm-card* ($S \cap T$) $\leq i$
**proof** –

**assume** *a1*: *vdm-card T* $\leq$ *i*
**assume** *a2*: *finite T*
**have** *f3*: $\forall$ *A Aa*. ((*card* (*A*::*'a set*) $\leq$ *card* (*Aa*::*'a set*) $\vee$ $\neg$ *vdm-card A* $\leq$ *vdm-card Aa*) $\vee$ *infinite A*) $\vee$ *infinite Aa*
**by** (*metis* (*full-types*) *l-vdm-card-finite of-nat-le-iff*)
**{ assume** *T* $\cap$ *S* $\neq$ *T*
**then have** *vdm-card* (*T* $\cap$ *S*) $\neq$ *vdm-card T* $\wedge$ *T* $\cap$ *S* $\neq$ *T* $\vee$ *vdm-card* (*T* $\cap$ *S*) $\leq$ *i*
**using** *a1* **by** *presburger*
**then have** *vdm-card* (*T* $\cap$ *S*) $\leq$ *i*
**using** *f3 a2 a1* **by** (*meson card-seteq dual-order.trans inf-le1 infinite-super verit-la-generic*) **}**
**then show** *?thesis*
**using** *a1* **by** (*metis* (*no-types*) *Int-commute*)
**qed**

**theorem** *l-vdm-card-fsb*:
*pre-vdm-card s* $\Longrightarrow$ *post-vdm-card s* (*vdm-card s*)
**by** (*simp add*: *inv-VDMNat-def inv-VDMSet-def post-vdm-card-def pre-vdm-card-def*)

@TODO power set

# 3 Sequences

**type-synonym** *'a VDMSeq* = *'a list*
**type-synonym** *'a VDMSeq1* = *'a list*

**definition**
*inv-VDMSeq1* :: *'a VDMSeq1* $\Rightarrow \mathbb{B}$
**where**
[*intro*!]: *inv-VDMSeq1 s* $\equiv$ *s* $\neq$ []

Sequences may have invariants within their inner type.

**definition**
*inv-SeqElems* :: (*'a* $\Rightarrow \mathbb{B}$) $\Rightarrow$ *'a VDMSeq* $\Rightarrow \mathbb{B}$
**where**
[*intro*!]: *inv-SeqElems einv s* $\equiv$ *list-all einv s*

**definition**
*inv-SeqElems0* :: (*'a* $\Rightarrow \mathbb{B}$) $\Rightarrow$ *'a VDMSeq* $\Rightarrow \mathbb{B}$
**where**
*inv-SeqElems0 einv s* $\equiv \forall$ *e* $\in$ (*set s*) . *einv e*

Isabelle's list *hd* and *tl* functions have the same name as VDM. Nevertheless, their results is defined for empty lists. We need to rule them out.

**definition**
*inv-VDMSeq'* :: (*'a* $\Rightarrow \mathbb{B}$) $\Rightarrow$ *'a VDMSeq* $\Rightarrow \mathbb{B}$
**where**

[*intro*!]: *inv-VDMSeq′ einv s* ≡ *inv-SeqElems einv s*

**definition**
*inv-VDMSeq1′* :: ($'a \Rightarrow \mathbb{B}$) ⇒ *'a VDMSeq1* ⇒ $\mathbb{B}$
**where**
[*intro*!]: *inv-VDMSeq1′ einv s* ≡ *inv-VDMSeq′ einv s* ∧ *inv-VDMSeq1 s*

**lemmas** *inv-VDMSeq′-defs* = *inv-VDMSeq′-def inv-SeqElems-def*
**lemmas** *inv-VDMSeq1′-defs* = *inv-VDMSeq1′-def inv-VDMSeq′-defs inv-VDMSeq1-def*

## 3.1 Sequence operators specification

**definition**
*len* :: *'a VDMSeq* ⇒ *VDMNat*
**where**
[*intro*!]: *len l* ≡ *int* (*length l*)

**definition**
*post-len* :: *'a VDMSeq* ⇒ *VDMNat* ⇒ $\mathbb{B}$
**where**
*post-len s R* ≡ *inv-VDMNat R* ∧ ($s \neq []$ ⟶ *inv-VDMNat1 R*)

**definition**
*elems* :: *'a VDMSeq* ⇒ *'a VDMSet*
**where**
[*intro*!]: *elems l* ≡ *set l*

**definition**
*post-elems* :: *'a VDMSeq* ⇒ *'a VDMSet* ⇒ $\mathbb{B}$
**where**
*post-elems s R* ≡ $R \subseteq$ *set s*

Be careful with representation differences VDM lists are 1-based, whereas Isabelle list are 0-based. This function returns 0,1,2 for sequence [A, B, C] instead of 1,2,3

**definition**
*inds0* :: *'a VDMSeq* ⇒ *VDMNat VDMSet*
**where**
*inds0 l* ≡ {*0* ..< *len l*}

**definition**
*inds* :: *'a VDMSeq* ⇒ *VDMNat1 VDMSet*
**where**
[*intro*!]: *inds l* ≡ {*1* .. *len l*}

**definition**
*post-inds* :: *'a VDMSeq* ⇒ *VDMNat1 VDMSet* ⇒ $\mathbb{B}$
**where**
*post-inds l R* ≡ *finite R* ∧ (*len l*) = (*card R*)

**definition**
*inds-as-nat* :: $'a$ *VDMSeq* $\Rightarrow \mathbb{N}$ *set*
**where**
*inds-as-nat l* $\equiv$ {*1* .. *nat* (*len l*)}

*applyList* plays with $'a$ *option* type instead of *undefined*.

**definition**
*applyList* :: $'a$ *VDMSeq* $\Rightarrow \mathbb{N} \Rightarrow$ $'a$ *option*
**where**
*applyList l n* $\equiv$ (*if* ($n > 0 \wedge$ *int n* $\leq$ *len l*) *then*
    *Some*(*l* ! (*n* − (*1*::*nat*)))
  *else*
    *None*)

*applyVDMSeq* sticks with *undefined*.

**definition**
*applyVDMSeq* :: $'a$ *VDMSeq* $\Rightarrow$ *VDMNat1* $\Rightarrow$ $'a$ (**infixl** $ *100*)
**where**
*l* $ *n* $\equiv$ (*if* (*inv-VDMNat1 n* $\wedge$ *n* $\leq$ *len l*) *then*
    (*l* ! *nat* (*n* − *1*))
  *else*
    *undefined*)

**definition**
*applyVDMSubseq'* :: $'a$ *VDMSeq* $\Rightarrow$ *VDMNat1* $\Rightarrow$ *VDMNat1* $\Rightarrow$ $'a$ *VDMSeq* (-$$$ (*1*{-..-})) **where**
*s* $$$ {*l*..*u*} $\equiv$ *if inv-VDMNat1 l* $\wedge$ *inv-VDMNat1 u* $\wedge$ (*l* $\leq$ *u*) *then*
    *nths s* {(*nat l*)−*1*..(*nat u*)−*1*}
  *else*
    []

Thanks to Tom Hayle for suggesting a generalised version, which is similar to the one below

**definition**
*applyVDMSubseq* :: $'a$ *VDMSeq* $\Rightarrow$ *VDMInt VDMSet* $\Rightarrow$ $'a$ *VDMSeq* (**infixl** $$ *105*)
**where**
*xs* $$ *s* $\equiv$ *nths xs* {*x*::*nat* | *x* . *x*+*1* $\in$ *s* }

**lemma** *l-vdm-len-fsb*: *post-len s* (*len s*)
**using** *post-len-def len-def*
**by** (*simp add*: *len-def post-len-def inv-VDMNat1-def inv-VDMNat-def*)

**lemma** *l-vdm-elems-fsb*: *post-elems s* (*elems s*)
**by** (*simp add*: *elems-def post-elems-def*)

**lemma** *l-vdm-inds-fsb*: *post-inds s* (*inds s*)
**using** *post-inds-def inds-def len-def*

**by** (*simp add*: *inds-def len-def post-inds-def*)

**lemma** *l-vdmsubseq-empty*[*simp*]:
$[] \ \$\$ \ \{l..u\} = []$ **unfolding** *applyVDMSubseq-def* **by** *simp*

**lemma** *l-vdmsubseq-beyond*[*simp*]:
$l > u \Longrightarrow s \ \$\$ \ \{l..u\} = []$ **unfolding** *applyVDMSubseq-def* **by** *simp*

**lemma** $len\ (s \ \$\$ \ \{i..j\}) = (min\ j\ ((len\ s) - (max\ 1\ i))) + 1$
**unfolding** *applyVDMSubseq-def len-def*
**apply** (*simp add*: *length-nths*)
**unfolding** *min-def max-def* **apply** (*simp*, *safe*)
**apply** (*induct s*)
**apply** *simp*
**apply** (*induct i* )
**oops**

**lemma** *l-vdmsubseq-ext-eq*:
$inv\text{-}VDMNat1\ l \Longrightarrow inv\text{-}VDMNat1\ u \Longrightarrow s \ \$\$\$ \ \{l..u\} = s \ \$\$ \ \{l..u\}$
**unfolding** *applyVDMSubseq-def applyVDMSubseq′-def inv-VDMNat1-def*
**apply** (*simp*;*safe*)
**apply** (*subgoal-tac* $\{nat\ l - Suc\ 0..nat\ u - Suc\ 0\} = \{x.\ l \le int\ x + 1 \wedge int\ x + 1 \le u\}$)
**apply** (*erule HOL.subst*; *simp*)
**apply** (*safe*;*simp*)
**apply** *linarith*+
**apply** (*subgoal-tac* $\{x.\ l \le int\ x + 1 \wedge int\ x + 1 \le u\} = \{\}$)
**apply** (*erule ssubst*,*simp*)
**by** *auto*

**lemmas** *applyVDMSeq-defs* = *applyVDMSeq-def inv-VDMNat1-def len-def*

**definition**
*pre-applyVDMSeq* :: $'a\ VDMSeq \Rightarrow VDMNat1 \Rightarrow \mathbb{B}$
**where**
$pre\text{-}applyVDMSeq\ xs\ i \equiv inv\text{-}VDMNat1\ i \wedge i \le len\ xs$

**definition**
*post-applyVDMSeq* :: $'a\ VDMSeq \Rightarrow VDMNat1 \Rightarrow 'a \Rightarrow \mathbb{B}$
**where**
$post\text{-}applyVDMSeq\ xs\ i\ R \equiv pre\text{-}applyVDMSeq\ xs\ i \longrightarrow R = xs\ \$\ i$

**theorem** *PO-applyVDMSeq-fsb*:
$\forall\ xs\ i\ .\ pre\text{-}applyVDMSeq\ xs\ i \longrightarrow post\text{-}applyVDMSeq\ xs\ i\ (xs\$i)$
**unfolding** *post-applyVDMSeq-def pre-applyVDMSeq-def* **by** *simp*

**definition**

$pre\text{-}applyVDMSubseq :: 'a\ VDMSeq \Rightarrow VDMNat1 \Rightarrow VDMNat1 \Rightarrow \mathbb{B}$
**where**
$pre\text{-}applyVDMSubseq\ xs\ l\ u \equiv inv\text{-}VDMNat1\ l \wedge inv\text{-}VDMNat1\ u \wedge l \leq u$

**definition**
$post\text{-}applyVDMSubseq :: 'a\ VDMSeq \Rightarrow VDMNat1 \Rightarrow VDMNat1 \Rightarrow 'a\ VDMSeq \Rightarrow \mathbb{B}$
**where**
$post\text{-}applyVDMSubseq\ xs\ l\ u\ R \equiv R = (if\ pre\text{-}applyVDMSubseq\ xs\ l\ u\ then\ (xs\$\$\{l..u\})\ else\ [])$

**theorem** *PO-applyVDMSubseq-fsb*:
$\forall\ xs\ i\ .\ pre\text{-}applyVDMSubseq\ xs\ l\ u \longrightarrow post\text{-}applyVDMSubseq\ xs\ l\ u\ (xs\$\$\{l..u\})$

**unfolding** *post-applyVDMSubseq-def pre-applyVDMSubseq-def* **by** *simp*

**definition**
$post\text{-}append :: 'a\ VDMSeq \Rightarrow 'a\ VDMSeq \Rightarrow 'a\ VDMSeq \Rightarrow \mathbb{B}$
**where**
$post\text{-}append\ s\ t\ r \equiv r = s\ @\ t$

**lemmas** *VDMSeq-defs* = *elems-def inds-def applyVDMSeq-defs*

**lemma** *l-applyVDMSeq-inds*[*simp*]:
$pre\text{-}applyVDMSeq\ xs\ i = (i \in inds\ xs)$
**unfolding** *pre-applyVDMSeq-def inv-VDMNat1-def len-def inds-def*
**by** *auto*

Isabelle *hd* and *tl* is the same as VDM

**definition**
$pre\text{-}hd :: 'a\ VDMSeq \Rightarrow \mathbb{B}$
**where**
$pre\text{-}hd\ s \equiv s \neq []$

**definition**
$post\text{-}hd :: 'a\ VDMSeq \Rightarrow 'a \Rightarrow \mathbb{B}$
**where**
$post\text{-}hd\ s\ RESULT \equiv pre\text{-}hd\ s \longrightarrow (RESULT \in elems\ s \vee RESULT = s\$1)$

**definition**
$pre\text{-}tl :: 'a\ VDMSeq \Rightarrow \mathbb{B}$
**where**
$pre\text{-}tl\ s \equiv s \neq []$

**definition**
$post\text{-}tl :: 'a\ VDMSeq \Rightarrow 'a\ VDMSeq \Rightarrow \mathbb{B}$
**where**
$post\text{-}tl\ s\ RESULT \equiv pre\text{-}tl\ s \longrightarrow elems\ RESULT \subseteq elems\ s$

**definition**
  *vdm-reverse* :: $'a$ *VDMSeq* $\Rightarrow$ $'a$ *VDMSeq*
  **where**
  [*intro*!]: *vdm-reverse xs* $\equiv$ *rev xs*

**definition**
  *post-vdm-reverse* :: $'a$ *VDMSeq* $\Rightarrow$ $'a$ *VDMSeq* $\Rightarrow \mathbb{B}$
  **where**
  *post-vdm-reverse xs R* $\equiv$ *elems xs* $=$ *elems R*

**definition**
  *conc* :: $'a$ *VDMSeq VDMSeq* $\Rightarrow$ $'a$ *VDMSeq*
  **where**
  [*intro*!]: *conc xs* $\equiv$ *concat xs*

**definition**
  *vdmtake* :: *VDMNat* $\Rightarrow$ $'a$ *VDMSeq* $\Rightarrow$ $'a$ *VDMSeq*
  **where**
  *vdmtake n s* $\equiv$ (*if inv-VDMNat n then take* (*nat n*) *s else* [])

**definition**
  *post-vdmtake* :: *VDMNat* $\Rightarrow$ $'a$ *VDMSeq* $\Rightarrow$ $'a$ *VDMSeq* $\Rightarrow \mathbb{B}$
  **where**
  *post-vdmtake n s RESULT* $\equiv$
    *len RESULT* $=$ *min n* (*len s*)
  $\wedge$ *elems RESULT* $\subseteq$ *elems s*

**definition**
  *seq-prefix* :: $'a$ *VDMSeq* $\Rightarrow$ $'a$ *VDMSeq* $\Rightarrow \mathbb{B}$ ((-/ $\sqsubseteq$ -) [*51*, *51*] *50*)
  **where**
  $s \sqsubseteq t \equiv (s = t) \vee (s = []) \vee (\mathit{len}\ s \leq \mathit{len}\ t \wedge (\exists\ i \in \mathit{inds}\ t\ .\ s = \mathit{vdmtake}\ i\ t))$

**definition**
  *post-seq-prefix* :: $'a$ *VDMSeq* $\Rightarrow$ $'a$ *VDMSeq* $\Rightarrow \mathbb{B} \Rightarrow \mathbb{B}$
  **where**
  *post-seq-prefix s t RESULT* $\equiv$
    *RESULT* $\longrightarrow$ (*elems s* $\subseteq$ *elems t* $\wedge$ *len s* $\leq$ *len t*)

## 3.2 Sequence operators lemmas

**lemma** *l-inv-VDMSet-finite*[*simp*]:
  *finite xs* $\Longrightarrow$ *inv-VDMSet xs*
  **unfolding** *inv-VDMSet-def* **by** *simp*

**lemma** *l-inv-SeqElems-alt*: *inv-SeqElems einv s* $=$ *inv-SeqElems0 einv s*
**by** (*simp add*: *elems-def inv-SeqElems0-def inv-SeqElems-def list-all-iff*)

**lemma** *l-inv-SeqElems-empty*[*simp*]: *inv-SeqElems f* []
  **by** (*simp add*: *inv-SeqElems-def*)

**lemma** *l-inv-SeqElems-Cons*: (*inv-SeqElems f* (*a*#*s*)) = (*f a* ∧ (*inv-SeqElems f s*))
**unfolding** *inv-SeqElems-def elems-def* **by** *auto*

**lemma** *l-inv-SeqElems-Cons'*: *f a* ⟹ *inv-SeqElems f s* ⟹ *inv-SeqElems f* (*a*#*s*)
**by** (*simp add*: *l-inv-SeqElems-Cons*)

**lemma** *l-inv-SeqElems-append*: (*inv-SeqElems f* (*xs* @ [*x*])) = (*f x* ∧ (*inv-SeqElems f xs*))
**unfolding** *inv-SeqElems-def elems-def* **by** *auto*

**lemma** *l-inv-SeqElems-append'*: *f x* ⟹ *inv-SeqElems f xs* ⟹ *inv-SeqElems f* (*xs* @ [*x*])
**by** (*simp add*: *l-inv-SeqElems-append*)

**lemma** *l-invSeqElems-inv-True-True*[*simp*]: *inv-SeqElems inv-True r*
**by** (*metis inv-SeqElems0-def l-inv-SeqElems-alt l-inv-True-True*)

**lemma** *l-len-nat1*[*simp*]: *s* ≠ [] ⟹ *0* < *len s*
**unfolding** *len-def* **by** *simp*

**lemma** *l-len-append-single*[*simp*]: *len*(*xs* @ [*x*]) = *1* + *len xs*
**apply** (*induct xs*)
**apply** *simp-all*
**unfolding** *len-def* **by** *simp-all*

**lemma** *l-len-empty*[*simp*]: *len* [] = *0* **unfolding** *len-def* **by** *simp*

**lemma** *l-len-cons*[*simp*]: *len*(*x* # *xs*) = *1* + *len xs*
**apply** (*induct xs*)
**unfolding** *len-def* **by** *simp-all*

**lemma** *l-elems-append*[*simp*]: *elems* (*xs* @ [*x*]) = *insert x* (*elems xs*)
**unfolding** *elems-def* **by** *simp*

**lemma** *l-elems-cons*[*simp*]: *elems* (*x* # *xs*) = *insert x* (*elems xs*)
**unfolding** *elems-def* **by** *simp*

**lemma** *l-elems-empty*[*simp*]: *elems* [] = {} **unfolding** *elems-def* **by** *simp*

**lemma** *l-inj-seq*: *distinct s* ⟹ *nat* (*len s*) = *card* (*elems s*)
**by** (*induct s*) (*simp-all add*: *elems-def len-def*)

**lemma** *l-elems-finite*[*simp*]:
*finite* (*elems l*)
**by** (*simp add*: *elems-def*)

**lemma** *l-inds-append*[*simp*]: *inds* (*xs* @ [*x*]) = *insert* (*len* (*xs* @ [*x*])) (*inds xs*)

**unfolding** *inds-def*
**by** (*simp add*: *atLeastAtMostPlus1-int-conv len-def*)

**lemma** *l-inds-cons*[*simp*]: *inds* (*x* # *xs*) = {*1* .. (*len xs* + *1*)}
  **unfolding** *inds-def len-def*
  **by** *simp*

**lemma** *l-len-within-inds*[*simp*]: $s \neq [] \Longrightarrow$ *len s* $\in$ *inds s*
**unfolding** *len-def inds-def*
**apply** (*induct s*)
**by** *simp-all*

**lemma** *l-inds-empty*[*simp*]: *inds* [] = {}
  **unfolding** *inds-def len-def* **by** *simp*

**lemma** *l-inds-as-nat-append*: *inds-as-nat* (*xs* @ [*x*]) = *insert* (*length* (*xs* @ [*x*])) (*inds-as-nat xs*)
**unfolding** *inds-as-nat-def len-def* **by** *auto*

**lemma** *l-applyVDM-len1*: *s* \$ (*len s* + *1*) = *undefined*
  **unfolding** *applyVDMSeq-def len-def* **by** *simp*

**lemma** *l-applyVDM-zero*[*simp*]: *s* \$ *0* = *undefined*
  **unfolding** *applyVDMSeq-defs* **by** *simp*

**lemma** *l-applyVDM1*: (*x* # *xs*) \$ *1* = *x*
  **by** (*simp add*: *applyVDMSeq-defs*)

**lemma** *l-applyVDM2*: (*x* # *xs*) \$ *2* = *xs* \$ *1*
  **by** (*simp add*: *applyVDMSeq-defs*)

**lemma** *l-applyVDM1-gen*[*simp*]: $s \neq [] \Longrightarrow$ *s* \$ *1* = *s* ! *0*
  **by** (*induct s*, *simp-all add*: *applyVDMSeq-defs*)

**lemma** *l-applyVDMSeq-i*[*simp*]: *i* $\in$ *inds s* $\Longrightarrow$ *s* \$ *i* = *s* ! *nat*(*i* − *1*)
  **unfolding** *applyVDMSeq-defs inds-def* **by** *simp*

**lemma** *l-applyVDM-cons-gt1empty*: *i* > *1* $\Longrightarrow$ (*x* # []) \$ *i* = *undefined*
  **by** (*simp add*: *applyVDMSeq-defs*)

**lemma** *l-applyVDM-cons-gt1*: *len xs* > *0* $\Longrightarrow$ *i* > *1* $\Longrightarrow$ (*x* # *xs*) \$ *i* = *xs* \$ (*i* − *1*)
  **apply** (*simp add*: *applyVDMSeq-defs*)
  **apply** (*intro impI*)
  **apply** (*induct xs rule*: *length-induct*)
  **apply** *simp-all*
  **by** (*smt nat-1 nat-diff-distrib*)

**lemma** *l-applyVDMSeq-defined*: $s \neq [] \Longrightarrow$ *inv-SeqElems* $(\lambda\ x\ .\ x \neq$ *undefined*$)$ $s$
$\Longrightarrow$ $s$ \$ (*len* $s$) $\neq$ *undefined*
**unfolding** *applyVDMSeq-defs*
**apply** (*simp*)
**apply** (*cases nat* (*int* (*length s*) − *1*))
**apply** *simp-all*
**apply** (*cases s*)
**apply** *simp-all*
**unfolding** *inv-SeqElems-def*
**apply** *simp*
**by** (*simp add*: *list-all-length*)

**lemma** *l-applyVDMSeq-append-last*:
(*ms* @ [*m*]) \$ (*len* (*ms* @ [*m*])) = *m*
**unfolding** *applyVDMSeq-defs*
**by** (*simp*)

**lemma** *l-applyVDMSeq-cons-last*:
(*m* # *ms*) \$ (*len* (*m* # *ms*)) = (*if ms* = [] *then m else ms* \$ (*len ms*))
**apply** (*simp*)
**unfolding** *applyVDMSeq-defs*
**by** (*simp add*: *nat-diff-distrib′*)

**lemma** *l-inds-in-set*:
$i \in$ *inds* $s \Longrightarrow s\$i \in$ *set* $s$
**unfolding** *inds-def applyVDMSeq-def inv-VDMNat1-def len-def*
**apply** (*simp*,*safe*)
**by** (*simp*)

**lemma** *l-inv-SeqElems-inds-inv-T*:
*inv-SeqElems inv-T s* $\Longrightarrow i \in$ *inds* $s \Longrightarrow$ *inv-T* $(s\$i)$
**apply** (*simp add*: *l-inv-SeqElems-alt*)
**unfolding** *inv-SeqElems0-def*
**apply** (*erule-tac* $x$=$s\$i$ **in** *ballE*)
**apply** *simp*
**using** *l-inds-in-set* **by** *blast*

**lemma** *l-inv-SeqElems-all*:
*inv-SeqElems inv-T s* = ($\forall\ i \in$ *inds* $s$ . *inv-T* $(s\$i)$)
**unfolding** *inv-SeqElems-def*
**apply** (*simp add*: *list-all-length*)
**unfolding** *inds-def len-def*
**apply** (*safe*,*simp*, *safe*)
**apply** (*erule-tac* $x$=*nat*($i$−*1*) **in** *allE*)
**apply** *simp*
**apply** (*erule-tac* $x$=*int* $n$ + *1* **in** *ballE*)
**by** *simp*+

```
lemma l-inds-upto: (i ∈ inds s) = (i ∈ {1..len s})
  by (simp add: inds-def)

lemma l-vdmtake-take[simp]: vdmtake n s = take n s
  unfolding vdmtake-def inv-VDMNat-def
  by simp

lemma l-seq-prefix-append-empty[simp]: s ⊑ s @ []
  unfolding seq-prefix-def
  by simp

lemma l-seq-prefix-id[simp]: s ⊑ s
  unfolding seq-prefix-def
  by simp

lemma l-len-append[simp]: len s ≤ len (s @ t)
  apply (induct t)
  by (simp-all add: len-def)

lemma l-vdmtake-len[simp]: vdmtake (len s) s = s
  unfolding vdmtake-def len-def inv-VDMNat-def by simp

lemma l-vdmtake-len-append[simp]: vdmtake (len s) (s @ t) = s
  unfolding vdmtake-def len-def inv-VDMNat-def by simp

lemma l-vdmtake-append[simp]: vdmtake (len s + len t) (s @ t) = (s @ t)
  apply (induct t)
   apply simp-all
  unfolding vdmtake-def len-def inv-VDMNat-def
  by simp

value vdmtake (1 + len [a,b,c]) ([a,b,c] @ [a])

lemma l-seq-prefix-append[simp]: s ⊑ s @ t
  unfolding seq-prefix-def
  apply (induct t)
  apply simp+
  apply (elim disjE)
    apply (simp-all)
  apply (cases s, simp)
  apply (rule disjI2, rule disjI2)
   apply (rule-tac x=len s in bexI)
    apply (metis l-vdmtake-len-append)
  using l-len-within-inds apply blast
   by (metis (full-types) atLeastAtMost-iff inds-def l-len-append l-len-within-inds
l-vdmtake-len-append)

lemma l-elems-of-inds-of-nth:
```

$1 < j \Longrightarrow j < int\ (length\ s) \Longrightarrow s\ !\ nat\ (j - 1) \in set\ s$
**by** *simp*

**lemma** *l-elems-inds-found*:
$x \in set\ s \Longrightarrow (\exists\ i\ .\ i < length\ s \wedge s\ !\ i = x)$

**apply** (*induct s*)
**apply** *simp-all*
**apply** *safe*
**by** *auto*

**lemma** *l-elems-of-inds*:
$(x \in elems\ s) = (\exists\ j\ .\ j \in inds\ s \wedge (s\$j) = x)$
**unfolding** *elems-def inds-def*
**apply** (*rule iffI*)
**unfolding** *applyVDMSeq-def len-def*
**apply** (*frule l-elems-inds-found*)
**apply** *safe*
**apply** (*rule-tac x=int(i)+1* **in** *exI*)
**apply** (*simp add*: *inv-VDMNat1-def*)
**using** *inv-VDMNat1-def* **by** *fastforce*

# 4 Optional inner type invariant check

**definition**
$inv\text{-}Option :: ('a \Rightarrow \mathbb{B}) \Rightarrow 'a\ option \Rightarrow \mathbb{B}$
**where**
$[intro!]: inv\text{-}Option\ inv\text{-}type\ v \equiv v \neq None \longrightarrow inv\text{-}type\ (the\ v)$

**lemma** *l-inv-option-Some*[*simp*]:
$inv\text{-}Option\ inv\text{-}type\ (Some\ x) = inv\text{-}type\ x$
**unfolding** *inv-Option-def*
**by** *simp*

**lemma** *l-inv-option-None*[*simp*]:
*inv-Option inv-type None*
**unfolding** *inv-Option-def*
**by** *simp*

# 5 Maps

In Isabelle, VDM maps can be declared by the $\rightharpoonup$ operator (not $\Rightarrow$) (i.e. type 'right' and you will see the arrow on dropdown menu).

It represents a function to an optional result as follows:

VDM : map X to Y Isabelle: $X \rightharpoonup Y$

which is the same as

Isabelle: $X \Rightarrow Y\ option$

where an optional type is like using nil in VDM (map X to [Y]). That is, Isabele makes the map total by mapping everything outside the domain to None (or nil). In Isabelle

$datatype\ 'a\ option = None \mid Some\ 'a$

Some VDM functions for map domain/range restriction and filtering. You use some like <: and :>. The use of some of these functions is one reason that makes the use of maps a bit more demanding, but it works fine. Given these are new definitions, "apply auto" won't finish proofs as Isabelle needs to know more (lemmas) about the new operators.

**definition**
  $inv\text{-}Map :: ('a \Rightarrow \mathbb{B}) \Rightarrow ('b \Rightarrow \mathbb{B}) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow \mathbb{B}$
**where**
  $[intro!]$:
  $inv\text{-}Map\ inv\text{-}Dom\ inv\text{-}Rng\ m \equiv$
    $inv\text{-}VDMSet'\ inv\text{-}Dom\ (dom\ m) \wedge$
    $inv\text{-}VDMSet'\ inv\text{-}Rng\ (ran\ m)$

**definition**
  $inv\text{-}Map1 :: ('a \Rightarrow \mathbb{B}) \Rightarrow ('b \Rightarrow \mathbb{B}) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow \mathbb{B}$
  **where**
  $[intro!]$: $inv\text{-}Map1\ inv\text{-}Dom\ inv\text{-}Ran\ m \equiv$
   $inv\text{-}Map\ inv\text{-}Dom\ inv\text{-}Ran\ m \wedge m \neq Map.empty$

**definition**
  $inv\text{-}Inmap :: ('a \Rightarrow \mathbb{B}) \Rightarrow ('b \Rightarrow \mathbb{B}) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow \mathbb{B}$
  **where**
  $[intro!]$: $inv\text{-}Inmap\ inv\text{-}Dom\ inv\text{-}Ran\ m \equiv$
   $inv\text{-}Map\ inv\text{-}Dom\ inv\text{-}Ran\ m \wedge inj\ m$

**lemmas** $inv\text{-}Map\text{-}defs = inv\text{-}Map\text{-}def\ inv\text{-}VDMSet'\text{-}defs$
**lemmas** $inv\text{-}Map1\text{-}defs = inv\text{-}Map1\text{-}def\ inv\text{-}Map\text{-}defs$
**lemmas** $inv\text{-}Inmap\text{-}defs = inv\text{-}Inmap\text{-}def\ inv\text{-}Map\text{-}defs\ inj\text{-}def$

**definition**
  $rng :: ('a \rightharpoonup 'b) \Rightarrow 'b\ VDMSet$
  **where**
  $[intro!]$: $rng\ m \equiv ran\ m$

**lemmas** $rng\text{-}defs = rng\text{-}def\ ran\text{-}def$

**definition**
  $dagger :: ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b)$ (**infixl** $\dagger$ *100*)
**where**
  $[intro!]$: $f \dagger g \equiv f ++ g$

**definition**
$munion :: ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b)$ (**infixl** $\cup m$ *90*)
**where**
$[intro!]$: $f \cup m\ g \equiv (if\ dom\ f \cap dom\ g = \{\}\ then\ f \dagger g\ else\ undefined)$

**definition**
$dom\text{-}restr :: 'a\ set \Rightarrow ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b)$ (**infixr** $\triangleleft$ *110*)
**where**
$[intro!]$: $s \triangleleft m \equiv m\ |`\ s$

**definition**
$dom\text{-}antirestr :: 'a\ set \Rightarrow ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b)$ (**infixr** $-\triangleleft$ *110*)
**where**
$[intro!]$: $s -\triangleleft m \equiv (\lambda x.\ if\ x : s\ then\ None\ else\ m\ x)$

**definition**
$rng\text{-}restr :: ('a \rightharpoonup 'b) \Rightarrow 'b\ set \Rightarrow ('a \rightharpoonup 'b)$ (**infixl** $\triangleright$ *105*)
**where**
$[intro!]$: $m \triangleright s \equiv (\lambda x\ .\ if\ (\exists\ y.\ m\ x = Some\ y \wedge y \in s)\ then\ m\ x\ else\ None)$

**definition**
$rng\text{-}antirestr :: ('a \rightharpoonup 'b) \Rightarrow 'b\ set \Rightarrow ('a \rightharpoonup 'b)$ (**infixl** $\triangleright-$ *105*)
**where**
$[intro!]$: $m \triangleright- s \equiv (\lambda x\ .\ if\ (\exists\ y.\ m\ x = Some\ y \wedge y \in s)\ then\ None\ else\ m\ x)$

**definition**
$vdm\text{-}merge :: ('a \rightharpoonup 'b)\ VDMSet \Rightarrow ('a \rightharpoonup 'b)$
**where**
$vdm\text{-}merge\ mm \equiv undefined$

**definition**
$vdm\text{-}inverse :: ('a \rightharpoonup 'b) \Rightarrow ('b \rightharpoonup 'a)$
**where**
$vdm\text{-}inverse\ m \equiv undefined$

**definition**
$map\text{-}subset :: ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow ('b \Rightarrow 'b \Rightarrow \mathbb{B}) \Rightarrow \mathbb{B}$ (((-)/ $\subseteq_s$ (-)/, (-)) [*0*, *0*, *50*] *50*)
**where**
$(m_1 \subseteq_s m_2,\ subset\text{-}of) \longleftrightarrow (dom\ m_1 \subseteq dom\ m_2 \wedge (\forall\, a \in dom\ m_1.\ subset\text{-}of\ (the(m_1\ a))\ (the(m_2\ a))))$

Map application is just function application, but the result is an optional type, so it is up to the user to unpick the optional type with the *the* operator. It means we shouldn't get to undefined, rather than we are handling undefinedness. That's because the value is comparable (see next lemma). In effect, if we ever reach undefined it means we have some partial function application outside its domain somewhere within any rewriting chain. As one cannot reason about this value, it can be seen as a flag for an error to be avoided.

**definition**
  $\mathit{map\text{-}comp} :: ('b \rightharpoonup 'c) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'c)$ (**infixl** $\circ m$ $55$)
  **where**
  $f \circ m\ g \equiv (\lambda\ x\ .\ \mathit{if}\ x \in \mathit{dom}\ g\ \mathit{then}\ f\ (\mathit{the}\ (g\ x))\ \mathit{else}\ \mathit{None})$

**definition**
  $\mathit{map\text{-}compatible} :: ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow \mathbb{B}$
  **where**
  $\mathit{map\text{-}compatible}\ m1\ m2 \equiv (\forall\ \ a \in \mathit{dom}\ m1 \cap \mathit{dom}\ m2\ .\ m1\ a = m2\ a)$

## 5.1 Map comprehension

Isabelle maps are similar to VDMs, but with some significant differences worth observing.

If the filtering is not unique (i.e. result is not a function), then the *THE x. P x* expression might lead to (undefined) unexpected results. In Isabelle maps, repetitions is equivalent to overriding, so that $[1 \mapsto 2::'a,\ 1 \mapsto 3::'a]$ $1 = \mathit{Some}\ (3::'a)$.

In various VDMToolkit definitions, we default to *undefined* in case where the situation is out of hand, hence, proofs will fail, and users will know that *undefined* being reached means some earlier problem has occurred.

Type bound map comprehension cannot filter for type invariants, hence won't have *undefined* results. This corresponds to the VDMSL expression

```
{ domexpr(d) |-> rngexpr(d, r) | d:S, r: T & P(d, r) }
```

where the maplet expression can be just variables or functions over the domain/range input(s).

VDM also issues a proof obligation for type bound maps (i.e. avoid it please!) to ensure the resulting map is finite. Concretely, the example below generates the corresponding proof obligation:

```
 ex: () -> map nat to nat
ex() == { x+y |-> 10 | x: nat, y in set {4,5,6} \& x < 10 };

  exists finmap1: map nat to (map (nat1) to (nat1)) &
      forall x:nat, y in set {4, 5, 6} & (x < 10) =>
        exists findex2 in set dom finmap1 &
```

```
finmap1(findex2) = {(x + y) |-> 10}
```

**definition**
$mapCompTypeBound :: ('a \Rightarrow \mathbb{B}) \Rightarrow ('b \Rightarrow \mathbb{B}) \Rightarrow ('a \Rightarrow 'b \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b \Rightarrow \mathbb{B}) \Rightarrow ('a \rightharpoonup 'b)$
**where**
$mapCompTypeBound\ inv\text{-}S\ inv\text{-}T\ domexpr\ rngexpr\ pred \equiv$
$(\lambda\ dummy::'a\ .$
$if\ (\exists\ d\ r\ .\ inv\text{-}S\ d \wedge inv\text{-}T\ r \wedge dummy = domexpr\ d\ r \wedge r = rngexpr\ d\ r \wedge pred\ d\ r)\ then$
$Some\ (THE\ r\ .\ inv\text{-}T\ r \wedge (\exists\ d\ .\ dummy = domexpr\ d\ r \wedge r = rngexpr\ d\ r \wedge pred\ d\ r))$
$else$
$None$
$)$

**value** $[1::nat \mapsto 2::nat,\ 3 \mapsto 3]\ 10$

Set bound map comprehension can filter bound set for their elements invariants. This corresponds to the VDMSL expression

```
{ domexpr(d, r) |-> rngexpr(d, r) | d in set S, r in set T & pred(d, r) }
{ domexpr(d, r) | d in set S , r in set T & pred(d, r) }
{ rngexpr(d, r) | d in set S , r in set T & pred(d, r) }
domexpr: S * T -> S
rngexpr: S * T -> T
pred   : S * T -> bool
```

If the types of `domexpr` or `rngexpr` are different from `S` or `T` then this will not work! If the filtering is not unique (i.e. result is not a function), then the $THE\ x.\ P\ x$ expression might lead to (undefined) unexpected results. In Isabelle maps, repetitions is equivalent to overriding, so that $[1 \mapsto 2,\ 1 \mapsto 3]\ 1 = Some\ 3$.

**definition**
$mapCompSetBound :: 'a\ set \Rightarrow 'b\ set \Rightarrow ('a \Rightarrow \mathbb{B}) \Rightarrow ('b \Rightarrow \mathbb{B}) \Rightarrow ('a \Rightarrow 'b \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b \Rightarrow \mathbb{B}) \Rightarrow ('a \rightharpoonup 'b)$
**where**
$mapCompSetBound\ S\ T\ inv\text{-}S\ inv\text{-}T\ domexpr\ rngexpr\ pred \equiv$
$(\lambda\ dummy::'a\ .$
— In fact you have to check the *inv-Type* of domexpr and rngexpr!!!
$if\ inv\text{-}VDMSet'\ inv\text{-}S\ S \wedge inv\text{-}VDMSet'\ inv\text{-}T\ T\ then$
$if\ (\exists\ r \in T\ .\ \exists\ d \in S\ .\ dummy = domexpr\ d\ r \wedge r = rngexpr\ d\ r \wedge pred\ d\ r)\ then$
$Some\ (THE\ r\ .\ r \in T \wedge inv\text{-}T\ r \wedge (\exists\ d \in S\ .\ dummy = domexpr\ d\ r \wedge r = rngexpr\ d\ r \wedge pred\ d\ r))$

*else*
— This is for map application outside its domain error, VDMJ 4061
*None*
*else*
— This is for type invariant violation errors, VDMJ ????
*undefined*
)

Identity functions to be used for the dom/rng expression functions for the case they are variables.

**definition**
$domid :: 'a \Rightarrow 'b \Rightarrow 'a$
**where**
$domid \equiv (\lambda\ d\ .\ (\lambda\ r\ .\ d))$

**definition**
$rngid :: 'a \Rightarrow 'b \Rightarrow 'b$
**where**
$rngid \equiv (\lambda\ d\ .\ id)$

Constant function to be used for the dom expression function for the case they are constants.

**definition**
$domcnst :: 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'a$
**where**
$domcnst\ v \equiv (\lambda\ d\ .\ (\lambda\ r\ .\ v))$

Constant function to be used for the rng expression function for the case they are constants.

**definition**
$rngcnst :: 'b \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$
**where**
$rngcnst\ v \equiv (\lambda\ d\ .\ (\lambda\ r\ .\ v))$

**definition**
$truecnst :: 'a \Rightarrow 'b \Rightarrow \mathbb{B}$
**where**
$truecnst \equiv (\lambda\ d\ .\ inv\text{-}True)$

**definition**
$predcnst :: \mathbb{B} \Rightarrow 'a \Rightarrow 'b \Rightarrow \mathbb{B}$
**where**
$predcnst\ p \equiv (\lambda\ d\ .\ (\lambda\ r\ .\ p))$

**lemma** *domidI*[*simp*]: $domid\ d\ r = d$
**by** (*simp add*: *domid-def*)

**lemma** *rngidI*[*simp*]: $rngid\ d\ r = r$

by (*simp add*: *rngid-def*)

**lemma** *domcnstI*[*simp*]: $domcnst\ v\ d\ r = v$
by (*simp add*: *domcnst-def*)

**lemma** *rngcnstI*[*simp*]: $rngcnst\ v\ d\ r = v$
by (*simp add*: *rngcnst-def*)

**lemma** *predcnstI*[*simp*]: $predcnst\ v\ d\ r = v$
by (*simp add*: *predcnst-def*)

**lemma** *truecnstI*[*simp*]: $r \neq undefined \Longrightarrow truecnst\ d\ r$
by (*simp add*: *truecnst-def*)

**lemmas** *maplet-defs* = *domid-def rngid-def rngcnst-def id-def truecnst-def inv-True-def*
**lemmas** *mapCompSetBound-defs* = *mapCompSetBound-def inv-VDMSet′-def inv-VDMSet-def maplet-defs rng-defs*
**lemmas** *mapCompTypeBound-defs* = *mapCompTypeBound-def maplet-defs rng-defs*

# 6 Lambda types

Lambda definitions entail an implicit satisfiability proof obligation check as part of its type invariant checks.

Because Isabelle lambdas are always curried, we need to also take this into account. For example, `lambda x: nat, y: nat1 & x+y` will effectively become (+). Thus callers to this invariant check must account for such currying when using more than one parameter in lambdas. (i.e. call this as *inv-Lambda inv-Dom* (*inv-Lambda inv-Dom′ inv-Ran*) *l* assuming the right invariant checks for the type of x and y and the result are used.

**definition**
$inv\text{-}Lambda :: ('a \Rightarrow \mathbb{B}) \Rightarrow ('b \Rightarrow \mathbb{B}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow \mathbb{B}$
**where**
$inv\text{-}Lambda\ inv\text{-}Dom\ inv\text{-}Ran\ l \equiv (\forall\ d\ .\ inv\text{-}Dom\ d \longrightarrow inv\text{-}Ran\ (l\ d))$

**definition**
$inv\text{-}Lambda' :: ('a \Rightarrow \mathbb{B}) \Rightarrow ('b \Rightarrow \mathbb{B}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow \mathbb{B}$
**where**
$inv\text{-}Lambda'\ inv\text{-}Dom\ inv\text{-}Ran\ l\ d \equiv inv\text{-}Dom\ d \longrightarrow inv\text{-}Ran\ (l\ d)$

# 7 Is test and type coercions

## 7.1 Basic type coercions

**definition**
$is\text{-}VDMRealWhole :: VDMReal \Rightarrow \mathbb{B}$
**where**
$is\text{-}VDMRealWhole\ r \equiv r \geq 1 \wedge (r - real\text{-}of\text{-}int\ (vdm\text{-}narrow\text{-}real\ r)) = 0$

**definition**
$vdmint\text{-}of\text{-}real :: VDMReal \rightharpoonup VDMInt$
**where**
$vdmint\text{-}of\text{-}real\ r \equiv if\ is\text{-}VDMRealWhole\ r\ then\ Some\ (vdm\text{-}narrow\text{-}real\ r)\ else\ None$

**definition**
$is\text{-}VDMRatWhole :: VDMRat \Rightarrow \mathbb{B}$
**where**
$is\text{-}VDMRatWhole\ r \equiv r \geq 1 \wedge (r - rat\text{-}of\text{-}int\ (vdm\text{-}narrow\text{-}real\ r)) = 0$

**definition**
$vdmint\text{-}of\text{-}rat :: VDMRat \rightharpoonup VDMInt$
**where**
$vdmint\text{-}of\text{-}rat\ r \equiv if\ is\text{-}VDMRatWhole\ r\ then\ Some\ (vdm\text{-}narrow\text{-}real\ r)\ else\ None$

## 7.2 Structured type coercions

**type-synonym** $('a, 'b)\ VDMTypeCoercion = 'a \rightharpoonup 'b$

A total VDM type coercion is one where every element in the type space of interest is convertible under the given type coercion (e.g., set of real = 1,2,3 into set of nat is total; whereas set of real = 0.5,2,3 into set of nat is not total given 0.5 is not nat).

**definition**
$total\text{-}coercion :: 'a\ VDMSet \Rightarrow ('a, 'b)\ VDMTypeCoercion \Rightarrow \mathbb{B}$
**where**
$total\text{-}coercion\ space\ conv \equiv (\forall\ i \in space\ .\ conv\ i \neq None)$

To convert a VDM set s of type 'a into type 'b (e.g., set of real into set of nat), it must be possible to convert every element of s under given type coercion

**definition**
$vdmset\text{-}of\text{-}t :: ('a, 'b)\ VDMTypeCoercion \Rightarrow ('a\ VDMSet, 'b\ VDMSet)\ VDMTypeCoercion$
**where**
$vdmset\text{-}of\text{-}t\ conv \equiv$
$(\lambda\ x\ .\ if\ total\text{-}coercion\ x\ conv\ then$
$\quad Some\ \{\ the(conv\ i) \mid i\ .\ i \in x \wedge conv\ i \neq None\ \}$
$else$
$\quad None)$

To convert a VDM seq s of type 'a into type 'b (e.g., seq of real into seq of nat), it must be possible to convert every element of s under given type coercion

**definition**

$vdmseq\text{-}of\text{-}t$ :: $('a,\ 'b)$ $VDMTypeCoercion \Rightarrow ('a\ VDMSeq,\ 'b\ VDMSeq)$ $VDMTypeCoercion$
**where**
$vdmseq\text{-}of\text{-}t\ conv \equiv$
$(\lambda\ x\ .\ if\ total\text{-}coercion\ (elems\ x)\ conv\ then$
$\quad Some\ [\ the(conv\ i)\ .\ i \leftarrow x,\ conv\ i \neq None\ ]$
$else$
$\quad None)$

### 7.3 Is tests

"Successful" is expr test is simply a call to the test expression invariant

**definition**
$isTest :: 'a \Rightarrow ('a \Rightarrow \mathbb{B}) \Rightarrow \mathbb{B}$
**where**
$[intro!]$: $isTest\ x\ inv\text{-}X \equiv inv\text{-}X\ x$

**lemma** $l\text{-}isTestI[simp]$: $isTest\ x\ inv\text{-}X = inv\text{-}X\ x$
**by** $(simp\ add:\ isTest\text{-}def)$

Possibly failing is expr tests up to given type coercion

**definition**
$isTest' :: 'a \Rightarrow ('a,\ 'b)\ VDMTypeCoercion \Rightarrow ('b \Rightarrow \mathbb{B}) \Rightarrow \mathbb{B}$
**where**
$[intro!]$: $isTest'\ x\ conv\ inv\text{-}X \equiv$
$(case\ conv\ x\ of$
$\quad None \Rightarrow False$
$\mid Some\ x \Rightarrow inv\text{-}X\ x)$

## 8 Set operators lemmas

**lemma** $l\text{-}psubset\text{-}insert$: $x \notin S \Longrightarrow S \subset insert\ x\ S$
**by** *blast*

**lemma** $l\text{-}right\text{-}diff\text{-}left\text{-}dist$: $S - (T - U) = (S - T) \cup (S \cap U)$
**by** $(metis\ Diff\text{-}Compl\ Diff\text{-}Int\ diff\text{-}eq)$
**thm** *Diff-Compl*
*Diff-Int*
*diff-eq*

**lemma** $l\text{-}diff\text{-}un\text{-}not\text{-}equal$: $R \subset T \Longrightarrow T \subseteq S \Longrightarrow S - T \cup R \neq S$
**by** *auto*

## 9 Map operators lemmas

**lemma** $l\text{-}map\text{-}non\text{-}empty\text{-}has\text{-}elem\text{-}conv$:
$g \neq Map.empty \longleftrightarrow (\exists\ x\ .\ x \in dom\ g)$

**by** (*metis domIff*)

**lemma** *l-map-non-empty-dom-conv*:
$g \neq Map.empty \longleftrightarrow dom\ g \neq \{\}$
**by** (*metis dom-eq-empty-conv*)

**lemma** *l-map-non-empty-ran-conv*:
$g \neq Map.empty \longleftrightarrow ran\ g \neq \{\}$
**by** (*metis empty-iff equals0I*
*fun-upd-triv option.exhaust*
*ranI ran-restrictD restrict-complement-singleton-eq*)

**lemma** *l-finite-rng*:
*finite* (*dom m*) $\Longrightarrow$ *finite* (*rng m*)
**by** (*simp add*: *finite-ran rng-def*)

### 9.0.1 Domain restriction weakening lemmas [EXPERT]

**lemma** *l-dom-r-iff*: $dom(S \triangleleft g) = S \cap dom\ g$
**by** (*metis Int-commute dom-restr-def dom-restrict*)

**lemma** *l-dom-r-subset*: $(S \triangleleft g) \subseteq_m g$
**by** (*metis Int-iff dom-restr-def l-dom-r-iff map-le-def restrict-in*)

**lemma** *l-dom-r-accum*: $S \triangleleft (T \triangleleft g) = (S \cap T) \triangleleft g$
**by** (*metis Int-commute dom-restr-def restrict-restrict*)

**lemma** *l-dom-r-nothing*: $\{\} \triangleleft f = Map.empty$
**by** (*metis dom-restr-def restrict-map-to-empty*)

**lemma** *l-dom-r-empty*: $S \triangleleft Map.empty = Map.empty$
**by** (*metis dom-restr-def restrict-map-empty*)

**lemma** *l-dres-absorb*: $UNIV \triangleleft m = m$
**by** (*simp add*: *dom-restr-def map-le-antisym map-le-def*)

**lemma** *l-dom-r-nothing-empty*: $S = \{\} \Longrightarrow S \triangleleft f = Map.empty$
**by** (*metis l-dom-r-nothing*)

**lemma** *f-in-dom-r-apply-elem*: $x \in S \Longrightarrow ((S \triangleleft f)\ x) = (f\ x)$
**by** (*metis dom-restr-def restrict-in*)

**lemma** *f-in-dom-r-apply-the-elem*: $x \in dom\ f \Longrightarrow x \in S \Longrightarrow ((S \triangleleft f)\ x) =$

*Some*(*the*(*f x*))
**by** (*metis domIff f-in-dom-r-apply-elem option.collapse*)

**lemma** *l-dom-r-disjoint-weakening*: $A \cap B = \{\} \Longrightarrow dom(A \lhd f) \cap dom(B \lhd f) = \{\}$
**by** (*metis dom-restr-def dom-restrict inf-bot-right inf-left-commute restrict-restrict*)

**lemma** *l-dom-r-subseteq*: $S \subseteq dom\, f \Longrightarrow dom\ (S \lhd f) = S$ **unfolding** *dom-restr-def*
**by** (*metis Int-absorb1 dom-restrict*)

**lemma** *l-dom-r-dom-subseteq*: $(dom\ (\ S \lhd f)) \subseteq dom\ f$
**unfolding** *dom-restr-def* **by** *auto*

**lemma** *l-the-dom-r*: $x \in dom\ f \Longrightarrow x \in S \Longrightarrow the\ ((\ S \lhd f)\ x) = the\ (f\ x)$
**by** (*metis f-in-dom-r-apply-elem*)

**lemma** *l-in-dom-dom-r*: $x \in dom\ (S \lhd f) \Longrightarrow x \in S$
  **by** (*metis Int-iff l-dom-r-iff*)

**lemma** *l-dom-r-singleton*: $x \in dom\ f \Longrightarrow (\{x\} \lhd f) = [x \mapsto the\ (f\ x)]$
**unfolding** *dom-restr-def*
**by** *auto*

**lemma** *singleton-map-dom*:
**assumes** $dom\ f = \{x\}$ **shows** $f = [x \mapsto the\ (f\ x)]$
**proof** –
**from** *assms* **obtain** $y$ **where** $f = [x \mapsto y]$
  **by** (*metis dom-eq-singleton-conv*)
**then have** $y = the\ (f\ x)$
**by** (*metis fun-upd-same option.sel*)
**thus** *?thesis* **by** (*metis* $\langle f = [x \mapsto y]\rangle$)
**qed**

**lemma** *l-relimg-ran-subset*:
  $ran\ (S \lhd m) \subseteq ran\ m$
  **by** (*metis* (*full-types*) *dom-restr-def ranI ran-restrictD subsetI*)

**lemma** *f-in-relimg-ran*:
  $y \in ran\ (S \lhd m) \Longrightarrow y \in ran\ m$
  **by** (*meson l-relimg-ran-subset subsetCE*)

**lemmas** *restr-simps* = *l-dom-r-iff l-dom-r-accum l-dom-r-nothing l-dom-r-empty*
  *f-in-dom-r-apply-elem l-dom-r-disjoint-weakening l-dom-r-subseteq*
  *l-dom-r-dom-subseteq*

### 9.0.2 Domain anti restriction weakening lemmas [EXPERT]

**lemma** *f-in-dom-ar-subsume*: $l \in dom\ (S -\triangleleft f) \Longrightarrow\ l \in dom\ f$
**unfolding** *dom-antirestr-def*
**by** (*cases l*$\in$*S*, *auto*)

**lemma** *f-in-dom-ar-notelem*: $l \in dom\ (\{r\} -\triangleleft f) \Longrightarrow l \neq r$
**unfolding** *dom-antirestr-def*
**by** *auto*

**lemma** *f-in-dom-ar-the-subsume*:
$l \in dom\ (S -\triangleleft f) \Longrightarrow the\ ((S -\triangleleft f)\ l) = the\ (f\ l)$
**unfolding** *dom-antirestr-def*
**by** (*cases l*$\in$*S*, *auto*)

**lemma** *f-in-dom-ar-apply-subsume*:
$l \in dom\ (S -\triangleleft f) \Longrightarrow ((S -\triangleleft f)\ l) = (f\ l)$
**unfolding** *dom-antirestr-def*
**by** (*cases l*$\in$*S*, *auto*)

**lemma** *f-in-dom-ar-apply-not-elem*: $l \notin S \Longrightarrow (S -\triangleleft f)\ l = f\ l$
**by** (*metis dom-antirestr-def*)

**lemma** *f-dom-ar-subset-dom*:
$dom(S -\triangleleft f) \subseteq dom\ f$
**unfolding** *dom-antirestr-def dom-def*
**by** *auto*

**lemma** *l-dom-dom-ar*:
$dom(S -\triangleleft f) = dom\ f - S$
**unfolding** *dom-antirestr-def*
**by** (*smt Collect-cong domIff dom-def set-diff-eq*)

**lemma** *l-dom-ar-accum*:
$S -\triangleleft (T -\triangleleft f) = (S \cup T) -\triangleleft f$
**unfolding** *dom-antirestr-def*
**by** *auto*

**lemma** *l-dom-ar-nothing*:
$S \cap dom\ f = \{\} \Longrightarrow S -\triangleleft f = f$

**unfolding** *dom-antirestr-def*
**apply** (*simp add*: *fun-eq-iff*)
**by** (*metis disjoint-iff-not-equal domIff*)

**lemma** *l-dom-ar-empty-lhs*:
$\{\} -\triangleleft f = f$
**by** (*metis Int-empty-left l-dom-ar-nothing*)

**lemma** *l-dom-ar-empty-rhs*:
$S -\triangleleft Map.empty = Map.empty$
**by** (*metis Int-empty-right dom-empty l-dom-ar-nothing*)

**lemma** *l-dom-ar-everything*:
$dom\ f \subseteq S \Longrightarrow S -\triangleleft f = Map.empty$
**by** (*metis domIff dom-antirestr-def in-mono*)

**lemma** *l-map-dom-ar-subset*: $S -\triangleleft f \subseteq_m f$
**by** (*metis domIff dom-antirestr-def map-le-def*)

**lemma** *l-dom-ar-none*: $\{\} -\triangleleft f = f$
**unfolding** *dom-antirestr-def*
**by** (*simp add*: *fun-eq-iff*)

**lemma** *l-map-dom-ar-neq*: $S \subseteq dom\ f \Longrightarrow S \neq \{\} \Longrightarrow S -\triangleleft f \neq f$
**apply** (*subst fun-eq-iff*)
**apply** (*insert ex-in-conv*[*of S*])
**apply** *simp*
**apply** (*erule exE*)
**unfolding** *dom-antirestr-def*
**apply** (*rule exI*)
**apply** *simp*
**apply** (*intro impI conjI*)
**apply** *simp-all*
**by** (*metis domIff set-mp*)

**lemma** *l-dom-rres-same-map-weaken*:
$S = T \Longrightarrow (S -\triangleleft f) = (T -\triangleleft f)$ **by** *simp*

**lemma** *l-dom-ar-not-in-dom*:
**assumes** $*$: $x \notin dom\ f$

**shows** $x \notin dom\ (s -\triangleleft f)$
**by** (*metis* $*$ *domIff dom-antirestr-def*)

**lemma** *l-dom-ar-not-in-dom2*: $x \in F \Longrightarrow x \notin dom\ (F\ -\triangleleft f)$
**by** (*metis domIff dom-antirestr-def*)

**lemma** *l-dom-ar-notin-dom-or*: $x \notin dom\ f \vee x \in S \Longrightarrow x \notin dom\ (S -\triangleleft f)$
**by** (*metis Diff-iff l-dom-dom-ar*)

**lemma** *l-in-dom-ar*: $x \notin F \Longrightarrow x \in dom\ f \Longrightarrow x \in dom\ \ (F\ -\triangleleft f)$
**by** (*metis f-in-dom-ar-apply-not-elem domIff*)

**lemma** *l-Some-in-dom*:
  $f\ x = Some\ y \Longrightarrow x \in dom\ f$ **by** *auto*

**lemma** *l-dom-ar-insert*: $((insert\ x\ F) -\triangleleft f) = \{x\} -\triangleleft (F {-\triangleleft} f)$
**proof**
  **fix** *xa*
  **show** $(insert\ x\ F -\triangleleft f)\ xa = (\{x\} -\triangleleft F -\triangleleft f)\ xa$
  **apply** (*cases* $x{=}\ xa$)
  **apply** (*simp add*: *dom-antirestr-def*)
  **apply** (*cases* $xa{\in}F$)
  **apply** (*simp add*: *dom-antirestr-def*)
  **apply** (*subst f-in-dom-ar-apply-not-elem*)
  **apply** *simp*
  **apply** (*subst f-in-dom-ar-apply-not-elem*)
  **apply** *simp*
  **apply** (*subst f-in-dom-ar-apply-not-elem*)
  **apply** *simp*
  **apply** *simp*
  **done**
**qed**

**lemma** *l-dom-ar-absorb-singleton*: $x \in F \Longrightarrow (\{x\} -\triangleleft F -\triangleleft f) =(F -\triangleleft f)$
**by** (*metis l-dom-ar-insert insert-absorb*)

**lemma** *l-dom-ar-disjoint-weakening*:
  $dom\ f \cap Y = \{\} \Longrightarrow dom\ (X -\triangleleft f) \cap Y = \{\}$
 **by** (*metis Diff-Int-distrib2 empty-Diff l-dom-dom-ar*)

**lemma** *l-dom-ar-singletons-comm*: $\{x\}{-\triangleleft}\ \{y\} -\triangleleft f = \{y\}{-\triangleleft}\ \{x\} -\triangleleft f$

**by** (*metis l-dom-ar-insert insert-commute*)

**lemma** *l-dom-r-ar-set-minus*:
$S \triangleleft (T -\triangleleft\, m) = (S - T) \triangleleft m$
**find-theorems** - = - *name*:*HOL* *name*:*fun*
**apply** (*rule ext*)
**unfolding** *dom-restr-def dom-antirestr-def restrict-map-def*
**by** *simp*

**lemmas** *antirestr-simps* = *f-in-dom-ar-subsume f-in-dom-ar-notelem f-in-dom-ar-the-subsume f-in-dom-ar-apply-subsume f-in-dom-ar-apply-not-elem f-dom-ar-subset-dom l-dom-dom-ar l-dom-ar-accum l-dom-ar-nothing l-dom-ar-empty-lhs l-dom-ar-empty-rhs l-dom-ar-everything l-dom-ar-none l-dom-ar-not-in-dom l-dom-ar-not-in-dom2 l-dom-ar-notin-dom-or l-in-dom-ar l-dom-ar-disjoint-weakening*

### 9.0.3 Map override weakening lemmas [EXPERT]

**lemma** *l-dagger-assoc*:
$f \dagger (g \dagger h) = (f \dagger g) \dagger h$
**by** (*metis dagger-def map-add-assoc*)
**thm** *ext option.split fun-eq-iff*

**lemma** *l-dagger-apply*:
$(f \dagger g)\ x = (if\ x \in dom\ g\ then\ (g\ x)\ else\ (f\ x))$
**unfolding** *dagger-def*
**by** (*metis* (*full-types*) *map-add-dom-app-simps*(*1*) *map-add-dom-app-simps*(*3*))

**lemma** *l-dagger-dom*:
$dom(f \dagger g) = dom\ f \cup dom\ g$
**unfolding** *dagger-def*
**by** (*metis dom-map-add sup-commute*)

**lemma** *l-dagger-lhs-absorb*:
$dom\ f \subseteq dom\ g \Longrightarrow f \dagger g = g$
**apply** (*rule ext*)
**by**(*metis dagger-def l-dagger-apply map-add-dom-app-simps*(*2*) *set-rev-mp*)

**lemma** *l-dagger-lhs-absorb-ALT-PROOF*:
$dom\ f \subseteq dom\ g \Longrightarrow f \dagger g = g$
**apply** (*rule ext*)
**apply** (*simp add*: *l-dagger-apply*)
**apply** (*rule impI*)
**find-theorems** - $\notin$ - $\Longrightarrow$ - *name*:*Set*
**apply** (*drule contra-subsetD*)
**unfolding** *dom-def*
**by** (*simp-all*)

**lemma** *l-dagger-empty-lhs*:
$Map.empty \dagger f = f$
**by** (*metis dagger-def empty-map-add*)

**lemma** *l-dagger-empty-rhs*:
$f \dagger Map.empty = f$
**by** (*metis dagger-def map-add-empty*)

**lemma** *dagger-notemptyL*:
$f \neq Map.empty \Longrightarrow f \dagger g \neq Map.empty$ **by** (*metis dagger-def map-add-None*)

**lemma** *dagger-notemptyR*:
$g \neq Map.empty \Longrightarrow f \dagger g \neq Map.empty$ **by** (*metis dagger-def map-add-None*)

**lemma** *l-dagger-dom-ar-assoc*:
$S \cap dom\ g = \{\} \Longrightarrow (S -\!\!\lhd f) \dagger g = S -\!\!\lhd (f \dagger g)$
**apply** (*simp add*: *fun-eq-iff*)
**apply** (*simp add*: *l-dagger-apply*)
**apply** (*intro allI impI conjI*)
**unfolding** *dom-antirestr-def*
**apply** (*simp-all add*: *l-dagger-apply*)
**by** (*metis dom-antirestr-def l-dom-ar-nothing*)
**thm** *map-add-comm*

**lemma** *l-dagger-not-empty*:
$g \neq Map.empty \Longrightarrow f \dagger g \neq Map.empty$
**by** (*metis dagger-def map-add-None*)

**lemma** *in-dagger-domL*:
$x \in dom\ f \Longrightarrow x \in dom(f \dagger g)$
**by** (*metis dagger-def domIff map-add-None*)

**lemma** *in-dagger-domR*:
$x \in dom\ g \Longrightarrow x \in dom(f \dagger g)$
**by** (*metis dagger-def domIff map-add-None*)

**lemma** *the-dagger-dom-right*:
**assumes** $x \in dom\ g$
**shows** $the\ ((f \dagger g)\ x) = the\ (g\ x)$

**by** (*metis assms dagger-def map-add-dom-app-simps*(*1*))

**lemma** *the-dagger-dom-left*:
**assumes** $x \notin dom\ g$
**shows** $the\ ((f \dagger g)\ x) = the\ (f\ x)$
**by** (*metis assms dagger-def map-add-dom-app-simps*(*3*))

**lemma** *the-dagger-mapupd-dom*: $x{\neq}y \Longrightarrow (f \dagger [y \mapsto z])\ x = f\ x$
**by** (*metis dagger-def fun-upd-other map-add-empty map-add-upd*)

**lemma** *dagger-upd-dist*: $f \dagger fa(e \mapsto r) = (f \dagger fa)(e \mapsto r)$ **by** (*metis dagger-def map-add-upd*)

**lemma** *antirestr-then-dagger-notin*: $x \notin dom\ f \Longrightarrow \{x\} -\lhd (f \dagger [x \mapsto y]) = f$
**proof**
**fix** $z$
**assume** $x \notin dom\ f$
**show** $(\{x\} -\lhd (f \dagger [x \mapsto y]))\ z = f\ z$
**by** (*metis* ‹$x \notin dom\ f$› *domIff dom-antirestr-def fun-upd-other insertI1 l-dagger-apply singleton-iff*)
**qed**
**lemma** *antirestr-then-dagger*: $r{\in}\ dom\ f \Longrightarrow \{r\} -\lhd f \dagger [r \mapsto the\ (f\ r)] = f$
**proof**
**fix** $x$
**assume** $*$: $r{\in}dom\ f$
**show** $(\{r\} -\lhd f \dagger [r \mapsto the\ (f\ r)])\ x = f\ x$
**proof** (*subst l-dagger-apply*,*simp*,*intro conjI impI*)
**assume** $x{=}r$ **then show** $Some\ (the\ (f\ r)) = f\ r$ **using** $*$ **by** *auto*
**next**
**assume** $x \neq r$ **then show** $(\{r\} -\lhd f)\ x = f\ x$ **by** (*metis f-in-dom-ar-apply-not-elem singleton-iff*)
**qed**
**qed**

**lemma** *dagger-notin-right*: $x \notin dom\ g \Longrightarrow (f \dagger g)\ x = f\ x$
**by** (*metis l-dagger-apply*)

**lemma** *dagger-notin-left*: $x \notin dom\ f \Longrightarrow (f \dagger g)\ x = g\ x$
**by** (*metis dagger-def map-add-dom-app-simps*(*2*))

**lemma** *l-dagger-commute*: $dom\ f \cap dom\ g = \{\} \Longrightarrow f \dagger g = g \dagger f$
**unfolding** *dagger-def*
**apply** (*rule map-add-comm*)
**by** *simp*

**lemmas** *dagger-simps* = *l-dagger-assoc l-dagger-apply l-dagger-dom l-dagger-lhs-absorb* *l-dagger-empty-lhs l-dagger-empty-rhs dagger-notemptyL dagger-notemptyR l-dagger-not-empty in-dagger-domL in-dagger-domR the-dagger-dom-right the-dagger-dom-left the-dagger-mapupd-dom dagger-upd-dist antirestr-then-dagger-notin antirestr-then-dagger dagger-notin-right dagger-notin-left*

### 9.0.4 Map update weakening lemmas [EXPERT]

without the condition nitpick finds counter example

**lemma** *l-inmapupd-dom-iff*:
$l \neq x \Longrightarrow (l \in dom\ (f(x \mapsto y))) = (l \in dom\ f)$
**by** (*metis* (*full-types*) *domIff fun-upd-apply*)

**lemma** *l-inmapupd-dom*:
$l \in dom\ f \Longrightarrow l \in dom\ (f(x \mapsto y))$
**by** (*metis dom-fun-upd insert-iff option.distinct*(*1*))

**lemma** *l-dom-extend*:
$x \notin dom\ f \Longrightarrow\ dom\ (f1(x \mapsto y)) = dom\ f1 \cup \{x\}$
**by** *simp*

**lemma** *l-updatedom-eq*:
$x{=}l \Longrightarrow the\ ((f(x \mapsto the\ (f\ x) - s))\ l) = the\ (f\ l) - s$
**by** *auto*

**lemma** *l-updatedom-neq*:
$x{\neq}l \Longrightarrow the\ ((f(x \mapsto the\ (f\ x) - s))\ l) = the\ (f\ l)$
**by** *auto*

— A helper lemma to have map update when domain is updated
**lemma** *l-insertUpdSpec-aux*: $dom\ f = insert\ x\ F \Longrightarrow (f0 = (f\ |\text{`}\ F)) \Longrightarrow f = f0\ (x \mapsto the\ (f\ x))$
**proof** *auto*
**assume** *insert*: $dom\ f = insert\ x\ F$
**then have** $x \in dom\ f$ **by** *simp*
**then show** $f = (f\ |\text{`}\ F)(x \mapsto the\ (f\ x))$ **using** *insert*
**unfolding** *dom-def*
**apply** *simp*
**apply** (*rule ext*)
**apply** *auto*
**done**
**qed**

**lemma** *l-the-map-union-right*: $x \in dom\ g \Longrightarrow dom\ f \cap dom\ g = \{\} \Longrightarrow the\ ((f \cup m\ g)\ x) = the\ (g\ x)$
**by** (*metis l-dagger-apply munion-def*)

**lemma** *l-the-map-union-left*: $x \in dom\ f \Longrightarrow dom\ f \cap dom\ g = \{\} \Longrightarrow the\ ((f \cup m\ g)\ x) = the\ (f\ x)$

**by** (*metis l-dagger-apply l-dagger-commute munion-def*)

**lemma** *l-the-map-union*: $dom\ f \cap dom\ g = \{\} \Longrightarrow the\ ((f \cup m\ g)\ x) = (if\ x \in dom\ f\ then\ the\ (f\ x)\ else\ the\ (g\ x))$
**by** (*metis l-dagger-apply l-dagger-commute munion-def*)

**lemmas** *upd-simps* = *l-inmapupd-dom-iff l-inmapupd-dom l-dom-extend l-updatedom-eq l-updatedom-neq*

### 9.0.5 Map union (VDM-specific) weakening lemmas [EXPERT]

**lemma** *k-munion-map-upd-wd*:
$x \notin dom\ f \Longrightarrow dom\ f \cap dom\ [x \mapsto y] = \{\}$
**by** (*metis Int-empty-left Int-insert-left dom-eq-singleton-conv inf-commute*)

**lemma** *l-munion-apply*:
$dom\ f \cap dom\ g = \{\} \Longrightarrow (f \cup m\ g)\ x = (if\ x \in dom\ g\ then\ (g\ x)\ else\ (f\ x))$
**unfolding** *munion-def*
**by** (*simp add*: *l-dagger-apply*)

**lemma** *l-munion-dom*:
$dom\ f \cap dom\ g = \{\} \Longrightarrow dom(f \cup m\ g) = dom\ f \cup dom\ g$
**unfolding** *munion-def*
**by** (*simp add*: *l-dagger-dom*)

**lemma** *l-diff-union*: $(A - B) \cup C = (A \cup C) - (B - C)$
**by** (*metis Compl-Diff-eq Diff-eq Un-Int-distrib2*)

**lemma** *l-munion-ran*: $dom\ f \cap dom\ g = \{\} \Longrightarrow ran(f \cup m\ g) = ran\ f \cup ran\ g$
**apply** (*unfold munion-def*)
**apply** *simp*
**find-theorems** (- † -) = -

**apply** (*intro set-eqI iffI*)
**unfolding** *ran-def*
**thm** *l-dagger-apply*
**apply** (*simp-all add*: *l-dagger-apply split-ifs*)

**apply** *metis*
**by** (*metis Int-iff all-not-in-conv domIff option.distinct*(*1*))

**lemma** *b-dagger-munion-aux*:
$dom(dom\ g -\lhd f) \cap dom\ g = \{\}$
**apply** (*simp add*: *l-dom-dom-ar*)
**by** (*metis Diff-disjoint inf-commute*)

**lemma** *b-dagger-munion*:
$(f \dagger g) = (dom\ g -\lhd f) \cup m\ g$
**find-theorems** (*300*) - = (-::(- ⇒ -)) $-name$:*Predicate* $-name$:*Product* $-name$:*Quick* $-name$:*New* $-name$:*Record* $-name$:*Quotient* $-name$:*Hilbert* $-name$:*Nitpick* $-name$:*Random* $-name$:*Transitive* $-name$:*Sum-Type* $-name$:*DSeq* $-name$:*Datatype* $-name$:*Enum* $-name$:*Big* $-name$:*Code* $-name$:*Divides*
**thm** *fun-eq-iff*[*of* $f \dagger g$ $(dom\ g -\lhd f) \cup m\ g$]
**apply** (*simp add*: *fun-eq-iff*)
**apply** (*simp add*: *l-dagger-apply*)
**apply** (*cut-tac b-dagger-munion-aux*[*of g f*])
**apply** (*intro allI impI conjI*)
**apply** (*simp-all add*: *l-munion-apply*)
**unfolding** *dom-antirestr-def*
**by** *simp*

**lemma** *l-munion-assoc*:
$dom\ f \cap dom\ g = \{\} \Longrightarrow dom\ g \cap dom\ h = \{\} \Longrightarrow (f \cup m\ g) \cup m\ h = f \cup m\ (g \cup m\ h)$
**unfolding** *munion-def*
**apply** (*simp add*: *l-dagger-dom*)
**apply** (*intro conjI impI*)
**apply** (*metis l-dagger-assoc*)
**apply** (*simp-all add*: *disjoint-iff-not-equal*)
**apply** (*erule-tac* [*1*−] *bexE*)
**apply** *blast*
**apply** *blast*
**done**

**lemma** *l-munion-commute*:
$dom\ f \cap dom\ g = \{\} \Longrightarrow f \cup m\ g = g \cup m\ f$
**by** (*metis b-dagger-munion l-dagger-commute l-dom-ar-nothing munion-def*)

**lemma** *l-munion-subsume*:
$x \in dom\ f \Longrightarrow the(f\ x) = y \Longrightarrow f = (\{x\} -\lhd f) \cup m\ [x \mapsto y]$
**apply** (*subst fun-eq-iff*)
**apply** (*intro allI*)
**apply** (*subgoal-tac* $dom(\{x\} -\lhd f) \cap dom\ [x \mapsto y] = \{\}$)
**apply** (*simp add*: *l-munion-apply*)
**apply** (*metis domD dom-antirestr-def singletonE option.sel*)
**by** (*metis Diff-disjoint Int-commute dom-eq-singleton-conv l-dom-dom-ar*)*Perhaps add* $g \subseteq_m f$ *instead?* **lemma** *l-munion-subsumeG*:
$dom\ g \subseteq dom\ f \Longrightarrow \forall\, x \in dom\ g\ .\ f\ x = g\ x \Longrightarrow f = (dom\ g -\lhd f) \cup m\ g$

**unfolding** *munion-def*
**apply** (*subgoal-tac* $dom\ (dom\ g -\lhd f) \cap dom\ g = \{\}$)
**apply** *simp*

**apply** (*subst fun-eq-iff*)
**apply** (*rule allI*)
**apply** (*simp add*: *l-dagger-apply*)
**apply** (*intro conjI impI*)+
**unfolding** *dom-antirestr-def*
**apply** (*simp*)
**apply** (*fold dom-antirestr-def*)
**by** (*metis Diff-disjoint inf-commute l-dom-dom-ar*)

**lemma** *l-munion-dom-ar-assoc*:
$S \subseteq dom\ f \Longrightarrow dom\ f \cap dom\ g = \{\} \Longrightarrow (S -\triangleleft f) \cup m\ g = S -\triangleleft (f \cup m\ g)$
**unfolding** *munion-def*
**apply** (*subgoal-tac* $dom\ (S -\triangleleft f) \cap dom\ g = \{\}$)
**defer** *1*
**apply** (*metis Diff-Int-distrib2 empty-Diff l-dom-dom-ar*)
**apply** *simp*
**apply** (*rule l-dagger-dom-ar-assoc*)
**by** (*metis equalityE inf-mono subset-empty*)

**lemma** *l-munion-empty-rhs*:
$(f \cup m\ Map.empty) = f$
**unfolding** *munion-def*
**by** (*metis dom-empty inf-bot-right l-dagger-empty-rhs*)

**lemma** *l-munion-empty-lhs*:
$(Map.empty \cup m\ f) = f$
**unfolding** *munion-def*
**by** (*metis dom-empty inf-bot-left l-dagger-empty-lhs*)

**lemma** *k-finite-munion*:
$finite\ (dom\ f) \Longrightarrow finite(dom\ g) \Longrightarrow dom\ f \cap dom\ g = \{\} \Longrightarrow finite(dom(f \cup m\ g))$
**by** (*metis finite-Un l-munion-dom*)

**lemma** *l-munion-singleton-not-empty*:
$x \notin dom\ f \Longrightarrow f \cup m\ [x \mapsto y] \neq Map.empty$
**apply** (*cases* $f = Map.empty$)
**apply** (*metis l-munion-empty-lhs map-upd-nonempty*)
**unfolding** *munion-def*
**apply** *simp*
**by** (*metis dagger-def map-add-None*)

**lemma** *l-munion-empty-iff*:
$dom\ f \cap dom\ g = \{\} \Longrightarrow (f \cup m\ g = Map.empty) \longleftrightarrow (f = Map.empty \wedge g = Map.empty)$
**apply** (*rule iffI*)
**apply** (*simp only*: *dom-eq-empty-conv*[*symmetric*] *l-munion-dom*)
**apply** (*metis Un-empty*)
**by** (*simp add*: *l-munion-empty-lhs l-munion-empty-rhs*)

**lemma** *l-munion-dom-ar-singleton-subsume*:
$x \notin dom\ f \Longrightarrow \{x\} -\triangleleft (f \cup m\ [x \mapsto y]) = f$
**apply** (*subst fun-eq-iff*)
**apply** (*rule allI*)
**unfolding** *dom-antirestr-def*
**by** (*auto simp*: *l-munion-apply*)

**lemma** *l-munion-upd*: $dom\ f \cap dom\ [x \mapsto y] = \{\} \Longrightarrow f \cup m\ [x \mapsto y] = f(x \mapsto y)$
**unfolding** *munion-def*
**apply** *simp*
**by** (*metis dagger-def map-add-empty map-add-upd*)

**lemma** *munion-notemp-dagger*: $dom\ f \cap dom\ g = \{\} \Longrightarrow f \cup m\ g \neq Map.empty \Longrightarrow f \dagger g \neq Map.empty$
**by** (*metis munion-def*)

**lemma** *dagger-notemp-munion*: $dom\ f \cap dom\ g = \{\} \Longrightarrow f \dagger g \neq Map.empty \Longrightarrow f \cup m\ g \neq Map.empty$
**by** (*metis munion-def*)

**lemma** *munion-notempty-left*: $dom\ f \cap dom\ g = \{\} \Longrightarrow f \neq Map.empty \Longrightarrow f \cup m\ g \neq Map.empty$
**by** (*metis dagger-notemp-munion dagger-notemptyL*)

**lemma** *munion-notempty-right*: $dom\ f \cap dom\ g = \{\} \Longrightarrow g \neq Map.empty \Longrightarrow f \cup m\ g \neq Map.empty$
**by** (*metis dagger-notemp-munion dagger-notemptyR*)

**lemma** *unionm-in-dom-left*: $x \in dom\ (f \cup m\ g) \Longrightarrow (dom\ f \cap dom\ g) = \{\} \Longrightarrow x \notin dom\ g \Longrightarrow x \in dom\ f$
**by** (*simp add*: *l-munion-dom*)

**lemma** *unionm-in-dom-right*: $x \in dom\ (f \cup m\ g) \Longrightarrow (dom\ f \cap dom\ g) = \{\} \Longrightarrow x \notin dom\ f \Longrightarrow x \in dom\ g$
**by** (*simp add*: *l-munion-dom*)

**lemma** *unionm-notin-dom*: $x \notin dom\ f \Longrightarrow x \notin dom\ g \Longrightarrow (dom\ f \cap dom\ g) = \{\} \Longrightarrow x \notin dom\ (f \cup m\ g)$
**by** (*metis unionm-in-dom-right*)

**lemmas** *munion-simps* = *k-munion-map-upd-wd l-munion-apply l-munion-dom b-dagger-munion l-munion-subsume l-munion-subsumeG l-munion-dom-ar-assoc l-munion-empty-rhs l-munion-empty-lhs k-finite-munion l-munion-upd munion-notemp-dagger dagger-notemp-munion munion-notempty-left munion-notempty-right*

**lemmas** *vdm-simps* = *restr-simps antirestr-simps dagger-simps upd-simps munion-simps*

### 9.0.6 Map finiteness weakening lemmas [EXPERT]

— Need to have the lemma options, otherwise it fails somehow
**lemma** *finite-map-upd-induct* [*case-names empty insert*, *induct set*: *finite*]:
**assumes** *fin*: *finite* (*dom f*)
**and** *empty*: *P Map.empty*
**and** *insert*: $\bigwedge e\ r\ f.$ *finite* (*dom f*) $\Longrightarrow e \notin$ *dom f* $\Longrightarrow P\ f \Longrightarrow P\ (f(e \mapsto r))$
**shows** *P f* **using** *fin*
**proof** (*induct dom f arbitrary*: *f rule*:*finite-induct*) — arbitrary statement is a must in here, otherwise cannot prove it
**case** *empty* **then have** *dom f* = {} **by** *simp* — need to reverse to apply rules
**then have** *f* = *Map.empty* **by** *simp*
**thus** *?case* **by** (*simp add*: *assms*(*2*))
**next**
**case** (*insert x F*)
— Show that update of the domain means an update of the map
**assume** *domF*: *insert x F* = *dom f* **then have** *domFr*: *dom f* = *insert x F* **by** *simp*
**then obtain** *f0* **where** *f0Def*: *f0* = *f* |‘ *F* **by** *simp*
**with** *domF* **have** *domF0*: *F* = *dom f0* **by** *auto*
**with** *insert* **have** *finite* (*dom f0*) **and** $x \notin$ *dom f0* **and** *P f0* **by** *simp-all*
**then have** *PFUpd*: *P* (*f0*($x \mapsto$ *the* (*f x*)))
**by** (*simp add*: *assms*(*3*))
**from** *domFr f0Def* **have** *f* = *f0*($x \mapsto$ *the* (*f x*)) **by** (*auto intro*: *l-insertUpdSpec-aux*)
**with** *PFUpd* **show** *?case* **by** *simp*
**qed**

**lemma** *finiteRan*: *finite* (*dom f*) $\Longrightarrow$ *finite* (*ran f*)
**proof** (*induct rule*:*finite-map-upd-induct*)
**case** *empty* **thus** *?case* **by** *simp*
**next**
**case** (*insert e r f*) **then have** *ranIns*: *ran* (*f*($e \mapsto r$)) = *insert r* (*ran f*) **by** *auto*
**assume** *finite* (*ran f*) **then have** *finite* (*insert r* (*ran f*)) **by** (*intro finite.insertI*)
**thus** *?case* **apply** (*subst ranIns*)
**by** *simp*
**qed**

**lemma** *l-dom-r-finite*: *finite* (*dom f*) $\Longrightarrow$ *finite* (*dom* ( $S \lhd f$))
**apply** (*rule-tac B*=*dom f* **in** *finite-subset*)
**apply** (*simp add*: *l-dom-r-dom-subseteq*)
**apply** *assumption*
**done**

**lemma** *dagger-finite*: *finite* (*dom* *f*) $\Longrightarrow$ *finite* (*dom* *g*) $\Longrightarrow$ *finite* (*dom* (*f* † *g*))
**by** (*metis dagger-def dom-map-add finite-Un*)

**lemma** *finite-singleton*: *finite* (*dom* [*a* $\mapsto$ *b*])
**by** (*metis dom-eq-singleton-conv finite.emptyI finite-insert*)

**lemma** *not-in-dom-ar*: *finite* (*dom* *f*) $\Longrightarrow$ *s* $\cap$ *dom* *f* = {} $\Longrightarrow$ *dom* (*s* $-\!\!\lhd$ *f*) = *dom* *f*
**apply** (*induct rule*: *finite-map-upd-induct*)
**apply** (*unfold dom-antirestr-def*) **apply** *simp*
**by** (*metis IntI domIff empty-iff*)

**lemma** *not-in-dom-ar-2*: *finite* (*dom* *f*) $\Longrightarrow$ *s* $\cap$ *dom* *f* = {} $\Longrightarrow$ *dom* (*s* $-\!\!\lhd$ *f*) = *dom* *f*
**apply** (*subst set-eq-subset*)
**apply** (*rule conjI*)
**apply** (*rule-tac*[!] *subsetI*)
**apply** (*metis l-dom-ar-not-in-dom*)
**by** (*metis l-dom-ar-nothing*)

**lemma** *l-dom-ar-commute-quickspec*:
*S* $-\!\!\lhd$ (*T* $-\!\!\lhd$ *f*) = *T* $-\!\!\lhd$ (*S* $-\!\!\lhd$ *f*)
**by** (*metis l-dom-ar-accum sup-commute*)

**lemma** *l-dom-ar-same-subsume-quickspec*:
*S* $-\!\!\lhd$ (*S* $-\!\!\lhd$ *f*) = *S* $-\!\!\lhd$ *f*
**by** (*metis l-dom-ar-accum sup-idem*)

**lemma** *l-map-with-range-not-dom-empty*: *dom* *m* $\neq$ {} $\Longrightarrow$ *ran* *m* $\neq$ {}
**by** (*simp add*: *l-map-non-empty-ran-conv*)

**lemma** *l-map-dom-ran*: *dom* *f* = *A* $\Longrightarrow$ *x* $\in$ *A* $\Longrightarrow$ *f* *x* $\neq$ *None*
**by** *blast*

**definition**
*seqcomp* :: ('*a* $\Rightarrow$ '*a*) $\Rightarrow$ ('*a* $\Rightarrow$ '*a*) $\Rightarrow$ '*a* $\Rightarrow$ '*a* (((-)/ ;; (-)/, (-)) [*0*, *0*, *10*] *10*)
**where**
[*intro*!]: (*P* ;; *Q*, *bst*) $\equiv$ *let mst* = *P bst in* (*Q mst*)

**fun**
*seqcomps* :: ('*a* $\Rightarrow$ '*a*) *list* $\Rightarrow$ '*a* $\Rightarrow$ '*a*
**where**
[*intro*!]: *seqcomps* [] *bst* = *bst*
| [*intro*!]: *seqcomps* (*x*#*xs*) *bst* = *seqcomps xs* (*x bst*)

**definition**
*seqcomps'* :: $('a \Rightarrow 'a)$ *list* $\Rightarrow 'a \Rightarrow 'a$
**where**
[*intro*!]: *seqcomps' l bst* = *foldl* ($\lambda$ *b* . ($\lambda$ *a* . *a b*)) *bst l*

**lemma** *l-seq-comp-simp*[*simp*]: (*P* ;; *Q*, *bst*) = *Q* (*P bst*) **unfolding** *seqcomp-def* **by** *simp*

**lemma** *l-ranE-frule*:
*e* $\in$ *ran f* $\Longrightarrow \exists$ *x* . *f x* = *Some e*
**unfolding** *ran-def* **by** *safe*

**lemma** *l-ranE-frule'*:
*e* $\in$ *ran f* $\Longrightarrow \exists$ *x* . *e* = *the*(*f x*)
**by** (*metis l-ranE-frule option.sel*)

**lemma** *l-inv-MapTrue*:
*finite* (*dom m*) $\Longrightarrow$ *undefined* $\notin$ *dom m* $\Longrightarrow$ *undefined* $\notin$ *rng m* $\Longrightarrow$ *inv-Map inv-True inv-True m*
**by** (*simp add*: *finite-ran inv-Map-def inv-VDMSet'-def rng-def*)

**lemma** *l-invMap-domr-absorb*:
*inv-Map di ri m* $\Longrightarrow$ *inv-Map di ri* (*S* $\lhd$ *m*)
**unfolding** *inv-Map-def inv-VDMSet'-defs inv-VDMSet-def*
**by** (*metis* (*mono-tags*, *lifting*) *domIff f-in-dom-r-apply-elem f-in-relimg-ran finiteRan l-dom-r-finite l-in-dom-dom-r*)

**lemma** *l-inv-Map-on-dom*: *inv-Map inv-Dom inv-Ran m* $\Longrightarrow$ *inv-SetElems inv-Dom* (*dom m*)
**unfolding** *inv-Map-defs* **by** *auto*

**lemma** *l-inv-Map-on-ran*: *inv-Map inv-Dom inv-Ran m* $\Longrightarrow$ *inv-SetElems inv-Ran* (*ran m*)
**unfolding** *inv-Map-defs* **by** *auto*

**lemma** *l-invMap-di-absorb*:
*undefined* $\notin$ *dom m* $\Longrightarrow$ *undefined* $\notin$ *rng m* $\Longrightarrow$ *inv-Map di ri m* $\Longrightarrow$ *inv-Map inv-True ri m*
**by** (*simp add*: *inv-Map-def inv-VDMSet'-def*)

# 10 To tidy up or remove

**value** *vdm-narrow-real* (*4.5*::*VDMRat*)
**value** *vdm-narrow-real* (*4.5*::*VDMReal*)

**value** *7 div* ( *3*::$\mathbb{Z}$) = *2*

**value** $7$ *vdmdiv* $(\ 3{::}\mathbb{Z}) = \ 2$

**value** $-7$ *div* $(-3{::}\mathbb{Z}) = \ 2$
**value** $-7$ *vdmdiv* $(-3{::}\mathbb{Z}) = \ 2$

**value** $-7$ *div* $(\ 3{::}\mathbb{Z}) = -3$
**value** $-7$ *vdmdiv* $(\ 3{::}\mathbb{Z}) = -2$

**value** $7$ *div* $(-3{::}\mathbb{Z}) = -3$
**value** $7$ *vdmdiv* $(-3{::}\mathbb{Z}) = -2$

**value** $1$ *div* $(-2{::}\mathbb{Z}) = -1$
**value** $1$ *vdmdiv* $(-2{::}\mathbb{Z}) = \ 0$
**value** $-1$ *div* $(\ 2{::}\mathbb{Z}) = -1$
**value** $-1$ *vdmdiv* $(\ 2{::}\mathbb{Z}) = \ 0$

**value** $0$ *div* $(-3{::}\mathbb{Z}) = 0$
**value** $0$ *vdmdiv* $(-3{::}\mathbb{Z}) = 0$
**value** $0$ *div* $(\ 3{::}\mathbb{Z}) = 0$
**value** $0$ *vdmdiv* $(\ 3{::}\mathbb{Z}) = 0$

**value** $7$ *mod* $(\ 3{::}\mathbb{Z}) = \ 1$
**value** $7$ *vdmmod* $(\ 3{::}\mathbb{Z}) = \ 1$

**value** $-7$ *mod* $(-3{::}\mathbb{Z}) = -1$
**value** $-7$ *vdmmod* $(-3{::}\mathbb{Z}) = -1$

**value** $-7$ *mod* $(\ 3{::}\mathbb{Z}) = \ 2$
**value** $-7$ *vdmmod* $(\ 3{::}\mathbb{Z}) = \ 2$

**value** $7$ *mod* $(-3{::}\mathbb{Z}) = -2$
**value** $7$ *vdmmod* $(-3{::}\mathbb{Z}) = -2$

**value** $7$ *vdmmod* $(\ 3{::}\mathbb{Z}) = \ 1$
**value** $-7$ *vdmmod* $(-3{::}\mathbb{Z}) = -1$
**value** $-7$ *vdmmod* $(\ 3{::}\mathbb{Z}) = \ 2$
**value** $7$ *vdmmod* $(-3{::}\mathbb{Z}) = -2$

**value** $7$ *vdmrem* $(\ 3{::}\mathbb{Z}) = \ 1$
**value** $-7$ *vdmrem* $(-3{::}\mathbb{Z}) = -1$
**value** $-7$ *vdmrem* $(\ 3{::}\mathbb{Z}) = -1$
**value** $7$ *vdmrem* $(-3{::}\mathbb{Z}) = \ 1$

**value** *inds0* $[A,\ B,\ C]$
**value** *nths* $[1,2,(3{::}nat)]\ \{2..3\}$

**value** *nths* $[A,B,C,D]\ \{(nat\ (-1))..(nat\ (-4))\}$

**value** *nths* [*A*,*B*,*C*,*D*] {(*nat* (−*4*))..(*nat* (−*1*))}
**value** [*A*,*B*,*C*,*D*]$$${−*4*..−*1*}
**value** [*A*,*B*,*C*,*D*]$$${−*1*..−*4*}
**value** [*A*,*B*,*C*,*D*,*E*]$$${*4*..*1*}
**value** [*A*,*B*,*C*,*D*,*E*]$$${*1*..*5*}
**value** [*A*,*B*,*C*,*D*,*E*]$$${*2*..*5*}
**value** [*A*,*B*,*C*,*D*,*E*]$$${*1*..*3*}
**value** [*A*,*B*,*C*,*D*,*E*]$$${*0*..*2*}
**value** [*A*,*B*,*C*,*D*,*E*]$$${−*1*..*2*}
**value** [*A*,*B*,*C*,*D*,*E*]$$${−*10*..*20*}
**value** [*A*,*B*,*C*,*D*,*E*]$$${*2*..−*1*}
**value** [*A*,*B*,*C*,*D*,*E*]$$${*2*..*2*}
**value** [*A*,*B*,*C*,*D*,*E*]$$${*0*..*1*}
**value** *len* ([*A*,*B*,*C*,*D*,*E*]$$${*2*..*2*})
**value** *len* ([*A*]$$${*2*..*2*})
**value** *card* {(*2*::*int*)..*2*}
**value** [*A*,*B*,*C*,*D*,*E*]$$${*0*..*0*}
**find-theorems** *card* {-..-}

## 10.1 Set translations: enumeration, comprehension, ranges

**value** { *x*+*x* | *x* . *x* ∈ {(*1*::*nat*),*2*,*3*,*4*,*5*,*6*} }
**value** { *x*+*x* | *x* . *x* ∈ {(*1*::*nat*),*2*,*3*} }

**value** {*0*..(*2*::*int*)}
**value** {*0*..<(*3*::*int*)}
**value** {*0*<..<(*3*::*int*)}

## 10.2 Seq translations: enumeration, comprehension, ranges

**value** { [*A*,*B*,*C*] ! *i* | *i* . *i* ∈ {*0*,*1*,*2*} }
**value** { [*A*,*B*,*C*,*D*,*E*,*F*] ! *i* | *i* . *i* ∈ {*0*,*2*,*4*} }

**value** [*A*, *B*, *C*] ! *0*
**value** [*A*, *B*, *C*] ! *1*
**value** [*A*, *B*, *C*] ! *2*
**value** [*A*, *B*, *C*] ! *3*
**value** *nth* [*A*, *B*, *C*] *0*

**value** *applyList* [*A*, *B*] *0* — out of range
**value** *applyList* [*A*, *B*] *1*
**value** *applyList* [*A*, *B*] *2*
**value** *applyList* [*A*, *B*] *3* — out of range

**value** [*A*,*B*,*C*,*D*] $ *0*
**lemma** [*A*,*B*,*C*] $ *4* = *A* **unfolding** *applyVDMSeq-defs* **apply** *simp* **oops**
**lemma** [*A*,*B*,*C*] $ *1* = *A* **unfolding** *applyVDMSeq-defs* **apply** *simp* **done**

**value** $[a]\ \$\ (len\ [(a::nat)])$
**value** $[A,\ B]\ \$\ 0$ — out of range
**value** $[A,B]\$1$
**value** $[A,\ B]\$\ 1$
**value** $[A,\ B]\$\ 2$
**value** $[A,\ B]\$\ 3$ — out of range

**value** $\{\ [A,B,C]\ !\ i\ |\ i\ .\ i \in \{0,1,2\}\ \}$
**value** $[\ x\ .\ x \leftarrow [0,1,(2::int)]\ ]$
**value** $[\ x\ .\ x \leftarrow [0\ ..\ 3]\ ]$

**value** $len\ [A,\ B,\ C]$
**value** $elems\ [A,\ B,\ C,\ A,\ B]$
**value** $elems\ [(0::nat),\ 1,\ 2]$
**value** $inds\ [A,B,C]$
**value** $inds\text{-}as\text{-}nat\ [A,B,C]$
**value** $card\ (elems\ [10,\ 20,\ 30,\ 1,\ 2,\ 3,\ 4,\ (5::nat),\ 10])$
**value** $len\ [10,\ 20,\ 30,\ 1,\ 2,\ 3,\ 4,\ (5::nat),\ 10]$

**type-synonym** $MySeq = VDMNat1\ list$
**definition**
$inv\text{-}MySeq :: MySeq \Rightarrow \mathbb{B}$
**where**
$inv\text{-}MySeq\ s \equiv (inv\text{-}SeqElems\ inv\text{-}VDMNat1\ s) \wedge$
$len\ s \leq 9 \wedge int\ (card\ (elems\ s)) = len\ s \wedge$
$(\forall\ i \in elems\ s\ .\ i > 0 \wedge i \leq 9)$

**value** $inv\text{-}MySeq\ [1,\ 2,\ 3]$

# 11 VDM PO layered expansion-proof strategy setup

I use various theorem tags to step-wise expand-simplify VDM goals

**lemmas** [*VDM-basic-defs*] = *inv-True-def inv-VDMChar-def*
*inv-VDMToken′-def inv-VDMToken-def*

**lemmas** [*VDM-num-defs*] = *inv-VDMNat-def inv-VDMNat1-def inv-VDMInt-def*
*inv-VDMReal-def inv-VDMRat-def*

**lemmas** [*VDM-num-fcns*] = *vdm-narrow-real-def vdm-div-def vdm-mod-def*
*vdm-rem-def vdm-pow-def vdm-abs-def vdm-floor-def*

**lemmas** [*VDM-num-spec-pre*] = *pre-vdm-mod-def pre-vdm-div-def*
*pre-vdm-rem-def pre-vdm-pow-def*

**lemmas** [*VDM-num-spec-post*] = *post-vdm-mod-def post-vdm-div-def*

*post-vdm-rem-def post-vdm-pow-def*
*post-vdm-floor-def post-vdm-abs-def*

**lemmas** [*VDM-set-defs*] = *inv-VDMSet-def inv-VDMSet1-def inv-VDMSet′-def inv-VDMSet1′-def inv-SetElems-def*
**lemmas** [*VDM-set-fcns*] = *vdm-card-def*
**lemmas** [*VDM-set-spec-pre*] = *pre-vdm-card-def*
**lemmas** [*VDM-set-spec-post*] = *post-vdm-card-def*

**lemmas** [*VDM-seq-defs*] = *inv-VDMSeq′-def inv-VDMSeq1′-def inv-SeqElems-def*
**lemmas** [*VDM-seq-fcns-1*] = *len-def elems-def inds-def inds-as-nat-def*
**lemmas** [*VDM-seq-fcns-2*] = *vdm-reverse-def vdmtake-def seq-prefix-def*
**lemmas** [*VDM-seq-fcns-3*] = *applyVDMSeq-def applyVDMSubseq′-def applyVDMSubseq-def*
**lemmas** [*VDM-seq-spec-pre*] = *pre-hd-def pre-tl-def pre-applyVDMSeq-def pre-applyVDMSubseq-def*
**lemmas** [*VDM-seq-spec-post-1*] = *post-len-def post-elems-def post-inds-def post-hd-def post-tl-def*
**lemmas** [*VDM-seq-spec-post-2*] = *post-vdm-reverse-def post-vdmtake-def post-seq-prefix-def post-append-def*
**lemmas** [*VDM-seq-spec-post-3*] = *post-applyVDMSeq-def post-applyVDMSubseq-def*

**lemmas** [*VDM-map-defs*] = *inv-Option-def inv-Map1-def inv-Map-def inv-Inmap-def*
**lemmas** [*VDM-map-fcns-1*] = *rng-def dagger-def munion-def*
**lemmas** [*VDM-map-fcns-2*] = *dom-restr-def dom-antirestr-def rng-restr-def rng-antirestr-def*
**lemmas** [*VDM-map-fcns-3*] = *vdm-merge-def vdm-inverse-def map-subset-def*

**lemmas** [*VDM-map-fcns-4*] = *map-comp-def map-compatible-def*
**lemmas** [*VDM-map-fcns-1-simps*] = *dagger-simps upd-simps munion-simps*
**lemmas** [*VDM-map-fcns-2-simps*] = *restr-simps antirestr-simps*
**lemmas** [*VDM-map-comp-1*] = *maplet-defs*
**lemmas** [*VDM-map-comp-2*] = *mapCompSetBound-defs*
**lemmas** [*VDM-map-comp-3*] = *mapCompTypeBound-defs*
**lemmas** [*VDM-num-crc-1*] = *is-VDMRealWhole-def is-VDMRatWhole-def*
**lemmas** [*VDM-num-crc-2*] = *vdmint-of-real-def vdmint-of-rat-def*
**lemmas** [*VDM-num-crc-3*] = *total-coercion-def vdmset-of-t-def vdmseq-of-t-def isTest-def isTest′-def*
**lemmas** [*VDM-stms-defs*] = *seqcomp-def seqcomps.simps seqcomps′-def*

**lemmas** [*VDM-num-spec*] = *VDM-num-spec-pre VDM-num-spec-post*
**lemmas** [*VDM-set-spec*] = *VDM-set-spec-pre VDM-set-spec-post*
**lemmas** [*VDM-seq-spec-post*] = *VDM-seq-spec-post-3 VDM-seq-spec-post-2 VDM-seq-spec-post-1*
**lemmas** [*VDM-seq-spec*] = *VDM-seq-spec-pre VDM-seq-spec-post*

**lemmas** [*VDM-seq-fcns*] = *VDM-seq-fcns-3 VDM-seq-fcns-2 VDM-seq-fcns-1*
**lemmas** [*VDM-map-fcns*] = *VDM-map-fcns-4 VDM-map-fcns-3 VDM-map-fcns-2*

*VDM-map-fcns-1*
**lemmas** [*VDM-map-fcns-simps*] = *VDM-map-fcns-2-simps VDM-map-fcns-1-simps*
**lemmas** [*VDM-map-comp*] = *VDM-map-comp-3 VDM-map-comp-2 VDM-map-comp-1*
**lemmas** [*VDM-num-crc*] = *VDM-num-crc-3 VDM-num-crc-2 VDM-num-crc-1*

**lemmas** [*VDM-num*] = *VDM-num-defs VDM-num-fcns VDM-num-crc*
**lemmas** [*VDM-set*] = *VDM-seq-defs VDM-set-fcns*
**lemmas** [*VDM-seq*] = *VDM-seq-defs VDM-seq-fcns*
**lemmas** [*VDM-map*] = *VDM-map-defs VDM-map-fcns VDM-map-comp*
**lemmas** [*VDM-stms*] = *VDM-stms-defs*
**lemmas** [*VDM-spec*] = *VDM-num-spec VDM-set-spec VDM-seq-spec*
**lemmas** [*VDM-all*] = *VDM-basic-defs VDM-num VDM-set VDM-seq VDM-map VDM-stms*