A translation strategy for VDM recursive functions in Isabelle/HOL

Leo Freitas[1]

School of Computing, Newcastle University,
leo.freitas@newcastle.ac.uk

Abstract. This paper presents a translation strategy for a variety of VDM recursive functions into Isabelle/HOL. It extends an ecosystem of other VDM mathematical toolkit extensions, including a translation and proof environment for VDM in Isabelle/HOL [1].

Keywords: VSCode, VDM, Isabelle/HOL, Translation, Recursion

1 Introduction

This paper describes a translation strategy for a variety of recursive definitions from VDM to Isabelle/HOL. The strategy takes into account the differences in how termination and well-foundedness are represented in both formalisms.

It is an extension to a VDM to Isabelle/HOL translation strategy and implementation[2] as a plugin to VDMJ [?] and extension to VDM-VSCode [?].

2 Background

The VDM to Isabelle/HOL translator caters for a wide range of the VDM AST. It copes with all kinds of expressions, a variety VDM patterns, all VDM types, VDM imports and exports, VDM functions and specifications, VDM traces, and some of VDM state and operations. Even though not all kinds of VDM patterns are allowed, the translator copes with most, and where it does not, a corresponding equivalent is possible. Among the expressions, map comprehension is of note, given its (implicit) complexity.

One particular area we want to extend the translation into is the one over recursively defined functions. VDM requires the user to define a measure function to justify why recursion will terminate. It then generates proof obligations to ensure totality and termination.

ANYTHING ELSE? Relates work?

[1] https://github.com/leouk/VDM_Toolkit/
[2] https://github.com/leouk/VDM_Toolkit

3 VDM basic types in Isabelle

Isabelle represents natural numbers ($\mathbb{N}$) as a (data) type with two constructors (0::'a and Suc n), where all numbers are projections over such constructions (e.g. 3 = Suc (Suc (Suc 0)). Isabelle integers ($\mathbb{Z}$) are defined as quotient type involving two natural numbers. Like with integers, other Isabelle numeric types (e.g. rationals$\mathbb{Q}$, reals $\mathbb{R}$, etc.) are defined in terms of some involved natural number construction. Type conversions (or coercions) are then defined to allow users to jump between type spaces. Nevertheless, Isabelle has no implicit type widening rule for $\mathbb{N}$; instead, it takes the following convention: 0 − x = 0. For mixutre of$\mathbb{Z}$ and $\mathbb{N}$, explicit user-defined type coercions are needed (e.g. int 2 − 3 = − 1).

VDM expressions with basic-typed (nat, int) variables have specific type widening rules (i.e. even if both variables are nat, the result might be int). Therefore, our translation strategy considers VDM nat as the Isabelle type VDMNat, which is just a type synonym for $\mathbb{Z}$. This simplifies the translation process to Isabelle, such that no type coercions are necessary to encode all VDM type widenning rules.

4 Recursion in VDM and in Isabelle

Recursive definitions are pervasive in most VDM models. For example, it can be used to perform iterations over numbers, lists, sets, etc. An important aspect of every recursive definition is an argument that justifies its termination (i.e. otherwise, the recursion might go on in an infinite loop).

In VDM, this is defined using a recursive measure: it has the same input type signature as the recursive definition, and returns a nat, which must monotonically decrease at each recursive call, eventually reaching zero. This is how termination of recursive definitions are justified in VDM.

A simple example of VDM recursive definition is one for calculating the factorial of a given natural number

```
factorial: nat -> nat
factorial(n) == if n = 0 then 1 else n * factorial(n - 1)
-- For the measure below, VDMJ produces a measure function as:
-- measure_factorial: nat -> nat
-- measure_factorial(n) == n
measure n;
```

The VDM recursive measure simply uses n itself; this works because the only recursive call is made with a decreasing value of n, until it reaches 0 and terminates. VDMJ generates three proof obligations for the definition above.

```
Proof Obligation 1: (Unproved)
factorial; measure_factorial: total function obligation in 'RecursiveVDM' at line 10:12
(forall n:nat & is_(measure_factorial(n), nat))
```

```
Proof Obligation 2: (Unproved)
factorial: subtype obligation in 'RecursiveVDM' at line 6:57
(forall n:nat & (not (n = 0) => (n - 1) >= 0))

Proof Obligation 3: (Unproved)
factorial: recursive function obligation in 'RecursiveVDM' at line 5:4
(forall n:nat & (not (n = 0) => measure_factorial(n) > measure_factorial((n - 1))))
```

They are trivial to discharge in Isabelle given the definition of measure_factorial is just n (e.g. $\forall$ n. n $\neq$ 0 $\longrightarrow$ 0 $\leq$ n $-$ 1, $\forall$ n. n $\neq$ 0 $\longrightarrow$ n $-$ 1 $<$ n).

In Isabelle, recursive definitions can be provided through primitive recursion or function definitions. The former insists on definition for each type constructor and only provides simplification rules; whereas the latter allow for more sophisticated input patterns and provides simplification, elimination and induction rules, as well as partial function considerations. For the purposes of this paper, we only consider function definitions. Readers can find more about these differences in [?].

Isabelle recursive functions requires a proof obligation that parameters represent a constructive and compatible pattern, and that recursive calls terminate. Constructive patterns relates to all constructors in data type being defined in the recursion (e.g., one equation for each of the constructors of $\mathbb{N}$, hence one involving 0::'a and another involving Suc n). Compatible patterns relates to multiple ways patterns can be constructed will boil down to the pattern completeness cases (e.g., n + (2::'a) being simply multiple calls over defined constructors like Suc (Suc n)). That is important to ensure that recursion is well structured (i.e., recursive calls will not get stuck because some constructs are not available). For example, if you miss the 0::'a case, eventually the Suc n case will reach zero and fail as no patterns for zero exist. The proof obligation for termination establishes that the recursion is well-founded. This has to be proved whenever properties of defined function are meant to be total.

Isabelle function definitions can be given with either fun or function. The former attempts to automatically prove the pattern constructive and compatible proofs and finds a measure for other termination proof obligations. The latter allows the user to do the proof manually, as well as provide a measure relation. This relation must be well-formed, which means have a well-ordered induction principle over a partially ordered relation[3]. For example, an Isabelle definition of factorial that it automatically discovers all three proofs can be given as

```
fun
  factorial :: ⟨ℕ ⇒ ℕ⟩
  where
  factorial n = (if n = 0 then 1 else n ∗ (factorial (n − 1)))
```

This Isabelle definition is pretty much 1-1 with the VDM definition. Nevertheless, as mentioned above, VDM basic types widening rules necessitated we translate

[3] For details on what that means in Isabelle, see the wellorder theorem wf {(x, y) | x < y} in theory Wellfounded.thy

them to VDMNat, which is just $\mathbb{Z}$. Moreover, $\mathbb{Z}$ is defined in terms of a pair of $\mathbb{N}$, hence recursion over $\mathbb{Z}$ will be involved. Thus, VDM recursive functions translation to Isabelle will be involved as well. For example, the same version of factorial defined for $\mathbb{Z}$ will fail with the error that Could not find lexicographic termination order. That is, Isabelle manages to discharge the pattern proofs $\mathbb{Z}$, but not the termination one.

Even if we could avoid this VDM basic types translation technicality, the same problem would occur for VDM recursion over non constructive types, such as sets or maps. That is, Isabelle only allow recursion over finite sets, which are not defined constructively but inductively (e.g. finite[display]. Similarly, Isabelle maps are defined with specialised HOL functions, again with domains that are not constructively defined. The only easy recursive definition translation from VDM to Isabelle are those involving lists, given lists in Isabelle are defined constructively and VDM sequences maps directly to them.

Therefore, defining recursive functions over non-constructive types entail more involved compatibility and completeness proofs. They also usually lead to partial function definitions, given Isabelle cannot tell whether termination is immediately obvious. In VDM, however, recursive functions on sets (as well as map domains) are common, hence the need for the extending the translation strategy.

5 VDM recursion translation strategy

We want to identify a translation strategy that will cater for such issues described above not only for basic types, but also for sets, sequences, maps, etc. This is important to ensure that the translator will cater for most commonly used VDM recursion definition patterns.

As mentioned in [?,?], it is possible to define formal annotations (as comments), which VDMJ will process and make available for its plugins. For our general translation strategy, we create a new annotation called @IsaMeasure. It defines a user-provided well-founded measure relation that will participate in the Isabelle proofs of termination. For example, for the factorial function above, the user would have to write

```
factorial: nat -> nat
factorial(n) == if n = 0 then 1 else n * factorial(n-1)
--@IsaMeasure( { (n -1, n) | n : nat & n <> 0 } )
measure n;
```

This measure relation corresponds to the relationship between the recursive call (factorial(n-1)) and its defining equation (factorial(n)), where the filtering condition determines for which values of n should the relation refer to (e.g. non-zero values). More interesting measure relation examples are defined in Section 5.4.

During translation, the plugin will typecheck the @IsaMeasure annotation (i.e., it is a type correct relation over the function signature). Next, it will translate the

annotation and some automation lemmas as series of Isabelle definitions to be used during the proof of termination of translated VDM recursive functions. If no annotation is provided, following similar principles from Isabelle, the plugin will try to automatically infer what the measure relation should based on the structure of the recursive function definition. When this fails, the user is informed. Still, even if measure relation synthesis succeeds, the user still have to appropriately use it during Isabelle's termination proof.

In what follows, we will detail the translation strategy for each relevant VDM type. For details over the overall translation strategy, see examples in the distribution[4] and [?]. That is we impose various implicit VDM checks as explicit predicates. For example, VDM sets are always finite, and type invariants over set elements must hold for every element.

5.1 Recursion over VDM basic types (nat, int)

Following the general translation strategy [?], we first encode the implicit precondition of factorial that insists that the given parameter n is a VDMNat, alongside a list of defining constants that are useful for proof strategy synthesis.

```
definition
  pre-vdm-factorial :: ‹VDMNat ⇒ 𝔹›
  where
  ‹pre-vdm-factorial n ≡ inv-VDMNat n›

lemmas pre-vdm-factorial-defs = pre-vdm-factorial-def inv-VDMNat-def
```

Next, we define the factorial function through recursion. When the precondition fails, we return undefined, which is a term that cannot be reasoned with in Isabelle (i.e. it is a dead end). Otherwise, we define factorial pretty much as in the VDM definition.

The domintros tag tells Isabelle to generate domain predicates, in case this function is not total. Domain predicates are important to our strategy because of course every VDM function will be undefined, when applied outside its precondition. It also generates domain-predicate sensitive proof rules listed below.

```
function (domintros) vdm-factorial :: ‹VDMNat ⇒ VDMNat›
  where
  ‹vdm-factorial n =
    (if pre-vdm-factorial n then
      (if n = 0 then
        1
      else
        n * (vdm-factorial (n - 1))
      )
    else
      undefined)›
```

[4] https://github.com/leouk/VDM_Toolkit

The proof obligations for pattern compatibility and completeness are next (⋀a. Wellfounded.accp vdm-factorial-rel a ⟹ ∃!y. vdm-factorial-graph a y) &&& (⋀P x. (⋀n. x = n ⟹ P) ⟹ P) 1. ⋀P x. (⋀n. x = n ⟹ P) ⟹ P 2. ⋀n na. n = na ⟹ (if pre-vdm-factorial n then if n = 0 then 1 else n ∗ vdm-factorial-sumC (n − 1) else undefined) = (if pre-vdm-factorial na then if na = 0 then 1 else na ∗ vdm-factorial-sumC (na − 1) else undefined)[display]

They are discharged with the usual Isabelle proof strategy for simple recursive patterns with the pat-completeness method. In the general (see more complex recursive call cases below in Section 5.4), the user might have goals to discharge.

by (pat-completeness, auto)

Various theorems are made available, such as:

- Case analysis (⋀n. ?x = n ⟹ ?P) ⟹ ?P[display]
- Elimination rules (partial) ⟦vdm-factorial ?x = ?y; vdm-factorial-dom ?x; ⋀n. ⟦?x = n; ?y = (if pre-vdm-factorial n then if n = 0 then 1 else n ∗ vdm-factorial (n − 1) else undefined); vdm-factorial-dom n⟧ ⟹ ?P⟧ ⟹ ?P[display]
- Induction rules (partial) ⟦vdm-factorial-dom ?a0.0; ⋀n. ⟦vdm-factorial-dom n; ⟦pre-vdm-factorial n; n ≠ 0⟧ ⟹ ?P (n − 1)⟧ ⟹ ?P n⟧ ⟹ ?P ?a0.0[display]
- Simplification rules (partial) vdm-factorial-dom ?n ⟹ vdm-factorial ?n = (if pre-vdm-factorial ?n then if ?n = 0 then 1 else ?n ∗ vdm-factorial (?n − 1) else undefined)[display]

  Note the last two are partial, module a domain predicate vdm-factorial-dom, which represents a well-founded relation that ensures termination. That is, if the user does not want (or know how to) to prove termination, such domain predicates will follow every application of the factorial definition, hence imposing users the requirement that such well-founded relation is still missing. If/when the termination proof is discharged, these p-rules can be simplified into total rules that do not depend on a domain predicate, given a well-founded relation has been provided. Domain predicates will complicate user proofs, and also make proof strategy synthesis harder to figure out.

  Termination proof is discharged by establishing a well-founded relation associated with the function recursive call(s) with respect to its declaration. In our case, the @IsaMeasure annotation is translated as the Isabelle abbreviation. We also implicitly add the filter that the function precondition holds: this is important to ensure the termination proof never reaches the undefined case. The other filter comes from the negated test in the definition if-statement. More complex definitions will have more involved filters (see Section 5.4. We use abbreviation instead of definition to avoid needing to expand the defined term.

abbreviation
  vdm-factorial-wf :: ‹(VDMNat × VDMNat) set›
  where

vdm-factorial-wf ≡ { (n − 1, n) | n . pre-vdm-factorial n ∧ n ≠ 0 }

Given vdm-factorial is a simple (non-mutual, single call-site, easy measure relation choice) recursion, thankfully the setup is not as complex to establish well-foundedness. For recursions of this nature, we can piggyback on some Isabelle machinery to help prove well foundedness by using the terms:

1. gen-VDMNat-term[display]
2. int-ge-less-than[display]

The first term is defined in terms of the second, which is a subset of our well-formed relation vdm-factorial-wf. Isabelle has proofs about the term's well formedness int-ge-less-than

wf (int-ge-less-than ?d)[display]

Thus, making the proof our term being well founded trivial, and easily discovered with proof tools like sledgehammer. As part of the translation strategy, we then define (and automatically discover the proof of) the following lemma. This follows the strategy described in [?].

lemma l-vdm-factorial-term-wf: wf (gen-VDMNat-term vdm-factorial-wf)
by (simp add: wf-int-ge-less-than wf-Int1)

termination

Finally, we prove termination using the previously proved lemma using the relation for All vdm-factorial-dom 1. All vdm-factorial-dom[display] This simplifies the goal into well formedness of termination relation and that the precondition implies it, both of which are easily proved with simplification in this case.

apply (relation ‹(gen-VDMNat-term vdm-factorial-wf)›)

This transforms the mysterious/abstract domain predicate into two new subgoals 1. wf (gen-VDMNat-term vdm-factorial-wf) 2. ⋀n. ⟦pre-vdm-factorial n; n ≠ 0⟧ ⟹ (n − 1, n) ∈ gen-VDMNat-term vdm-factorial-wf[display]

using l-vdm-factorial-term-wf apply presburger
by (simp add: pre-vdm-factorial-defs int-ge-less-than-def)

For this simple example, these goals are proved with sledgehammer. In general, the user will be have to either find the proof, or deal with domain predicates in proofs involving the recursive call.

After the termination proof is discharged, Isabelle provides total versions of useful rules as:

- Elimination rules (total) ⟦vdm-factorial ?x = ?y; ⋀n. ⟦?x = n; ?y = (if pre-vdm-factorial n then if n = 0 then 1 else n ∗ vdm-factorial (n − 1) else undefined)⟧ ⟹ ?P⟧ ⟹ ?P[display]
- Induction rules (total) (⋀n. (⟦pre-vdm-factorial n; n ≠ 0⟧ ⟹ ?P (n − 1)) ⟹ ?P n) ⟹ ?P ?a0.0[display]

– Simplification rules (total) vdm-factorial ?n = (if pre-vdm-factorial ?n then if ?n = 0 then 1 else ?n ∗ vdm-factorial (?n − 1) else undefined)[display]

To make sure our choice is valid (e.g. doesn't lead to the empty relation), we ensure that indeed the termination relation is in fact the same as the well founded predicate by proving the next goal. This is something users might want to do, but is not part of the translation strategy. In case the measure relation is empty, the recursive call simplification rules will not be useful anyhow.

```
lemma l-vdm-factorial-term-valid: (gen-VDMNat-term vdm-factorial-wf) = vdm-factorial-wf
 apply (simp add: pre-vdm-factorial-defs)
 apply (intro equalityI subsetI)
  apply (simp-all add: int-ge-less-than-def case-prod-beta)
 by auto
```

5.2 Recursion over VDM sets

Next, we extend the translation strategy for basic types for VDM sets. For this, we will use a recursively defined VDM function over sets that sums the set elements as

```
sumset: set of nat -> nat
sumset(s) == if s = {} then 0 else let e in set s in sumset(s - {e}) + e
--@IsaMeasure( { (x - { let e in set x in e }, x) | x : set of nat & x <> {} } )
--@Witness(sumset({ 1 }))
measure card s;
```

Like most common VDM recursion over sets, the function consumes the set by picking each set element and them calling the recursive call without the element picked, until the set is empty. The VDM measure states that the recursion is based on the cardinality of the input parameter; this is not useful for Isabelle's recursive definition proofs and is ignored during translation.

In Isabelle, the implicit VDM checks are defined as the precondition, which ensures that the given set contains only natural numbers, and is finite, as defined by inv-VDMSet′[display]

```
definition
 pre-sumset :: VDMNat VDMSet ⇒ 𝔹
 where
 pre-sumset s ≡ inv-VDMSet′ inv-VDMNat s

lemmas pre-sumset-defs = pre-sumset-def inv-VDMSet′-defs inv-VDMNat-def
```

We define the VDM recursive function in Isabelle next. It checks whether the given set satisfy the function precondition, returning undefined if not. If it does, then each case is encoded pretty much 1-1 from VDM. The translation strategy for VDM let-in-set patterns uses Isabelle's Hilbert's choice operator (SOME x. x ∈ s). Note this naturally extends to VDM's let-be-st patterns as well.

```
function (domintros)
  sumset :: VDMNat VDMSet ⇒ VDMNat
  where
  sumset s =
    (if pre-sumset s then
        (if s = {} then
            0
         else
           let e = (ϵ x . x ∈ s) in
             sumset (s − {e}) + e)
     else
        undefined
    )
```

The pattern completeness and compatibility goals are given as (⋀a. Wellfounded.accp sumset-rel a ⟹ ∃!y. sumset-graph a y) &&& (⋀P x. (⋀s. x = s ⟹ P) ⟹ P) 1. ⋀P x. (⋀s. x = s ⟹ P) ⟹ P 2. ⋀s sa. s = sa ⟹ (if pre-sumset s then if s = ∅ then 0 else let e = SOME x. x ∈ s in sumset-sumC (s − {e}) + e else undefined) = (if pre-sumset sa then if sa = ∅ then 0 else let e = SOME x. x ∈ sa in sumset-sumC (sa − {e}) + e else undefined)[display]

We follow the usual proof strategy for this using pat-completeness. For more general examples, if that fails, sledgehammer should be used.

```
  by (pat-completeness, auto)
```

The measure relation for termination is defined with the @IsaMeasure annotation above as the smaller set after picking e (e.g. s − {SOME e. e ∈ s}), and the set used at the entry call, leading to the pairs (s − {SOME e. e ∈ s}, s). Finally, we ensure all the relation elements satisfy the function precondition (pre-sumset), and that the if-test is negated.

```
abbreviation
  sumset-wf-rel :: (VDMNat VDMSet × VDMNat VDMSet) set
  where
  sumset-wf-rel ≡ { (s − {(ϵ e . e ∈ s)}, s)| s . pre-sumset s ∧ s ≠ {}}
```

Given this is a simple (non-mutual, single call-site, easy set element choice) recursion, again we can piggyback on Isabelle machinery by using the terms:

1. gen-set-term[display]
2. finite-psubset[display]
   They establishes that a relation where the first element is strictly smaller set than the second element in the relation pair is well-formed. This makes the proof of well-foundedness easy for sumset-wf-rel through sledgehammer.

```
lemma l-sumset-rel-wf: wf (gen-set-term sumset-wf-rel)
  using l-gen-set-term-wf by blast

termination
```

Next, we tackle the termination proof, with the same setup with relation again.

```
apply (rule termination[of (gen-set-term sumset-wf-rel)])
using l-sumset-rel-wf apply force
```

Unfortunately, using sledgehammer fails to discharge the second subgoal 1. ⋀s x. ⟦pre-sumset s; s ≠ ∅; x = (SOME x. x ∈ s)⟧ ⟹ (s − {x}, s) ∈ gen-set-term sumset-wf-rel[display]

```
oops
```

Fortunately, for most simple situations, this is easy to decompose in general. The translation strategy takes the @IsaMeasure expression and decompose its parts, such that the filtering predicates are assumptions, and the element in the relation belong to the well-formed measure chosen. For the concrete set example, this is defined in the next lemma, which require some manual intervention to tell Isabelle what definitions to unfold and simplify with. Then, Isabelle's sledgehammer can finish the proof.

```
lemma l-pre-sumset-sumset-wf-rel:
  pre-sumset s ⟹ s ≠ {} ⟹ (s − {(ϵ x. x ∈ s)}, s) ∈ (gen-set-term sumset-wf-rel)
  unfolding gen-set-term-def
  apply (simp add: pre-sumset-defs)
  by (metis Diff-subset member-remove psubsetI remove-def some-in-eq)
```

The intuition behind this lemma is that, elements in the measure relation satisfy well-formedness under the function precondition and the filtering case (s ≠ ∅) where the recursive call is made. That is, the precondition and filtering condition help establish the terminating relation. For this particular proof, the only aspect needed from the precondition (pre-sumset) is that the set is finite.

With this, we can try the termination proof again, which now sledgehammer find proofs for all subgoals.

```
termination
  apply (rule termination[of (gen-set-term sumset-wf-rel)])
  using l-sumset-rel-wf apply force
  using l-pre-sumset-sumset-wf-rel by presburger
```

Note we omit such lemma over termination and precondition for the VDMNat case in Section 5.1. The translation strategy does define it following the same recipe: recursive function precondition and filtering predicate as assumptions, and chosen termination relation element containment, where sledgehammer find the proof once more.

```
lemma l-pre-vdm-factorial-wf-rel: pre-vdm-factorial n ⟹ n ≠ 0 ⟹ (n − 1, n) ∈
gen-VDMNat-term vdm-factorial-wf
  unfolding gen-VDMNat-term-def gen-VDMInt-term-def
 using inv-VDMNat-def l-less-than-VDMNat-subset-int-ge-less-than pre-vdm-factorial-def
by auto
```

We also choose to show the relation finite subset trick to make well-founded induction proofs easier does not compromise the well founded relation itself.

```
lemma l-sumset-wf-rel-valid: gen-set-term sumset-wf-rel = sumset-wf-rel
 apply (intro equalityI subsetI)
  apply (simp)
 using l-pre-sumset-sumset-wf-rel by blast
```

Finally, even though this was not necessary for this proof, we encourage users to always provide a witness for the top recursive call. This is done by using the @Witness annotation [?,?]: it provides a concrete example for the function input parameters. This witness is useful for existentially quantified predicates present in more involved termination proofs (see Section 5.4).

5.3 Recursion over VDM maps

Recursive functions over VDM maps are a special case of VDM sets, given map recursion usually iterates over the map's domain. For example, the function that sums the elements of the map range can be defined as

```
sum_elems: map nat to nat -> nat
sum_elems(m) ==
   if m = {|->} then 0 else
     let d in set dom m in m(d) + sum_elems({d}<-:m)
--@IsaMeasure( { ({d} <-: m, m) | m : map nat to nat, d: nat & m <> {} and d in set dom m } )
--@Witness( sum_elems({ 1 |-> 1 }) )
measure card dom m;
```

As with sets, it iterates over the map by picking a domain element, performing the necessary computation, and then recurse on the map filtered by removing the chosen element, until the map is empty and the function terminates. As before, the measure relation follows the same pattern: recursive call site related with defining site, where both the if-test and the let-in choice is part of the filtering predicate.

Following the general translation strategy for maps [?], we define the function precondition using inv-Map[display] It insists that both the map domain and range are finite, and that all domain and range elements satisfy their corresponding type invariant. Note that if the recursion was defined over sets other than the domain and range, Isabelle might require you to prove such set is finite. Given both domain and range sets are themselves finite, this should be easy enough to do, if needed.

```
definition
 pre-sum-elems :: ‹(VDMNat ⇀ VDMNat) ⇒ 𝔹›
 where
 ‹pre-sum-elems m ≡ inv-Map inv-VDMNat inv-VDMNat m›

lemmas pre-sum-elems-defs = pre-sum-elems-def inv-Map-defs inv-VDMNat-def
```

VDM maps in Isabelle (VDMNat ⇀ VDMNat) are defined as a HOL function which maps to an optional result. That is, if the element is in the domain, the map results in a non nil value, whereas if the element does not belong to the domain, the map results in a nil value. This effectively makes all maps total, where values outside the domain map to nil. The Isabelle translation and compatibility proof follows patterns used before and are given as

```
function (domintros)
  sum-elems :: ‹(VDMNat ⇀ VDMNat) ⇒ VDMNat›
  where
  ‹sum-elems m =
      (if pre-sum-elems m then
          (if m = Map.empty then
            0
          else
            let d = (ϵ e . e ∈ dom m) in the(m d) + (sum-elems ({d} −◁ m))
          )
       else
          undefined
      )
  ›
  by (pat-completeness, auto)
```

Similarly, the well-formed relation from @IsaMeasure is translated next, where the precondition is also included as part of the relation's filter. The {d} −◁ m corresponds to VDM domain anti-filtering operator <-:.

```
abbreviation
  sum-elems-wf :: ‹((VDMNat ⇀ VDMNat) × (VDMNat ⇀ VDMNat)) VDMSet›
  where
  ‹sum-elems-wf ≡ { (({d} −◁ m), m) | m d . pre-sum-elems m ∧ m ≠ Map.empty ∧
d ∈ dom m }›
```

For the well-formed lemma over the recursive measure relation, there are no available Isabelle help, and projecting the domain element of the maps within the relation is awkward. Thus, we have to prove the lemma directly. This will not be automatic in general. This is one difference in terms of translation of VDM recursive functions over sets and maps.

Fortunately, the proof strategy for such situations is somewhat known: it follows a similar strategy to the proof of well formedness of the finite-psubset as wf finite-psubset[display]. The proof uses the VDM measure expression to extract the right projection of interest, then follows the proof for finite-psubset. Finally, sledgehammer can figure out the final steps.

```
lemma l-sum-elems-wf: wf sum-elems-wf
  apply (rule wf-measure[of ‹λ m . card (dom m)›, THEN wf-subset])
  apply (simp add: measure-def inv-image-def less-than-def less-eq)
  apply (rule subsetI, simp add: case-prod-beta)
  apply (elim exE conjE)
  by (simp add: l-VDMMap-filtering-card pre-sum-elems-defs)
```

The precondition subgoal and the termination proof follow the same patterns as before. Again, their proof was discovered with sledgehammer, yet this will not be the case in general.

```
lemma l-pre-sum-elems-sum-elems-wf:
  pre-sum-elems m ⟹ m ≠ Map.empty ⟹ ({(ϵ e. e ∈ dom m)} −⩤ m, m) ∈
sum-elems-wf
 apply (simp add: pre-sum-elems-defs)
 by (metis domIff empty-iff some-in-eq)

termination
 apply (rule termination[OF l-sum-elems-wf])
 using l-pre-sum-elems-sum-elems-wf by presburger
```

Finally, we also prove that the well founded termination relation (sum-elems-wf) is not empty, as we did for sets and nat recursion. Note that here the @Witness annotation is useful in discharging the actual value to use as the witness demonstrating the relation is not empty.

```
lemma l-sum-elems-wf-valid: sum-elems-wf ≠ {}
 apply safe
 apply (erule equalityE)
 apply (simp add: subset-eq)
 apply (erule-tac x=⟨[1 ↦ 1]⟩ in allE)
 by (simp add: pre-sum-elems-defs)
```

5.4 VDM recursion involving complex measures

```
ack: nat * nat -> nat
ack(m,n) ==
          if m = 0 then n+1
     else if n = 0 then ack(m-1, 1)
     else               ack(m-1, ack(m, (n-1)))
--@IsaMeasure( pair_less_VDMNat )
--@Witness( ack(2, 1) )
measure is not yet specified;
```

The plugin has examples of a couple other more complicated recursion. These are Nipkow's permutation function [?] and Takeuchi's function[5].

```
perm: int * int * int -> int --nat
perm(m,n,r) ==
          if 0 < r then perm(m, r-1, n)
     else if 0 < n then perm(r, n-1, m) else m
pre ((0 < r or 0 < n) => m+n+r > 0)
measure maxs({m+n+r, 0});

tak: int * int * int -> int
tak(x,y,z) ==
     if x <= y then y
     else           tak(tak(x-1,y,z), tak(y-1,z,x), tak(z-1,x,y))
measure is not yet specified;
```

[5] https://isabelle.in.tum.de/library/HOL/HOL-Examples/Functions.html

The Takeuchi's function is particularly challenging in proofs and the translation strategy stands no chance of finding proofs for such definitions automatically.

6 Limitations

7 Discussion and conclusion

Future work. lemma l-sumset-rel-wf′: wf sumset-wf-rel
apply (rule wf-measure[of ‹λ s . card s›, THEN wf-subset])
apply (simp add: measure-def inv-image-def less-than-def less-eq)
apply (rule subsetI, simp add: case-prod-beta)
apply (elim exE conjE)
by (metis card-Diff1-less-iff fst-conv inv-Map-defs(2) inv-Map-defs(3) pre-sumset-defs(1) snd-conv some-in-eq)