A translation strategy for VDM recursive functions in Isabelle/HOL

Leo Freitas, Peter Gorm Larsen

November 24, 2022

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 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.

Beyond overcoming technical practicalities, which we discuss, a major objective is to create translation strategy templates. These templates must cover a wide variety of VDM recursive definitions, as well as having their proof obligations being highly automated. The result is an extension to a VDM to Isabelle/HOL translation strategy and implementation as a plugin to VDMJ [?] and extension to VDM-VSCode [?].

Isabelle uses literate programming, where formal specification, proofs and documentation are all within the same environment. We omit proofs scripts below; the full VDM and Isabelle sources and proofs can be found at the VDM toolkit repository at ./plugins/vdm2isa/src/main/resources/RecursiveVDM.*[1].

In the next section, we present background on VDM and Isabelle recursion and measure relations. In Section 3 we briefly discuss VDM basic types translation and their consequence for recursion. Next, Section 4 describes how both VDM and Isabelle recursive definitions work and how they differ. Our translation strategy is then presented in Section 5 for basic types, sets, maps, and complex recursive patterns. Finally, we conclude in Section 6.

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

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 of patterns, almost all types, imports and exports, functions and specifications, traces, and some of 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 complexity. Details can be found at [?, ?].

One particular area we want to extend translation is 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.

Finally, our translation strategy follows the size-change termination (SCT) proof strategy described in [?, ?]. In particular, its SCP (polynomial) and SCNP (non-polynomial) subclass of recursive definitions within the SCT, which permits efficient termination certificate checking. Effectively, if every infinite computation would give rise to an infinitely decreasing value sequence (according to the size-change principle), then no infinite computation is possible. Termination problems in this class have a global ranking function of a certain form, which can be found using SAT solving, hence increasing automation.

ANYTHING ELSE? Related work?

3 VDM basic types in Isabelle

Isabelle represents natural numbers ($\mathbb{N}$) as a (data) type with two constructors (0 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 a quotient type involving two natural numbers. Isabelle quotient types are injections into a constructively defined type. 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 conventions like $(0::\mathbb{N}) - (x::\mathbb{N}) = (0::\mathbb{N})$. For expressions involving a mixutre of $\mathbb{Z}$ and $\mathbb{N}$ typed terms, explicit user-defined type coercions are needed (e.g., int $(2::\mathbb{N})$ $- (3::\mathbb{Z}) = - (1::\mathbb{Z})$).

VDM expressions with basic-typed (nat, int) variables have specific type widening rules. For example, even if both variables are nat, the result might be int. (e.g., in VDM 0 - x:nat = -x: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. On the other hand, this design decision means encoding of recursive functions over nat to be more complicated than expected, given VDM nat is represented as Isabelle's $\mathbb{Z}$.

Despite this design decision over basic types and their consequences, recursion over VDM int, sets or maps will still be involved. That is because these types are not constructively defined in Isabelle.

4 Recursion in VDM and in Isabelle

Recursive definitions are pervasive in VDM models. An important aspect of every recursive definition is an argument that justifies its termination. 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 the n input 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 at line 10:12
(forall n:nat & is_(measure_factorial(n), nat))

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

Proof Obligation 3: (Unproved)
factorial: recursive function obligation 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 measure definition ex-

panded is just $\forall$ n. n $\neq$ 0 $\longrightarrow$ 0 $\leq$ n $-$ 1 and $\forall$ n. n $\neq$ 0 $\longrightarrow$ n $-$ 1 $<$ n.

Moreover, even though measures over recursive type structures are impossible to define in VDM, they are easily described in Isabelle. For example, it is not possible to write a measure in VDM for a recursive function over a recursive record defining a linked list, such as R :: v: nat n: R. This is automatically generated for our representation of VDM records in Isabelle as a datatype. Other complex recursive patterns are hard/impossible to represent in VDM (see Section ??).

In Isabelle, recursive definitions can be provided through primitive recursion over inputs are constructively defined, or more general function definitions that produces proof obligations. 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 used in the recursion inputs (e.g., one equation for each of the constructors of $\mathbb{N}$, hence one involving 0 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 being simply multiple calls over defined constructors like Suc (Suc n)). This 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 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 the defined function are meant to be total.

Isabelle function definitions can be given with either fun or function syntax. The former attempts to automatically prove the pattern constructive and compatible proofs and finds a measure for the termination proof obligation. The latter requires the user to do these proofs manually by providing a measure relation. It is better suited for cases where fun declarations fail, which usually involve complex or ill-defined recusion.

The termination relation must be well-formed, which means have a well-ordered induction principle over a partially ordered relation defined as

wf (?r::(?'a $\times$?'a) set) =
($\forall$ P::?'a $\Rightarrow$ $\mathbb{B}$.
  ($\forall$ x::?'a. ($\forall$ y::?'a. (y, x) $\in$?r $\longrightarrow$ P y) $\longrightarrow$ P x) $\longrightarrow$ ($\forall$ x::?'a. P x))

[2]. 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 them to VDMNat, which is just $\mathbb{Z}$. 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 for $\mathbb{Z}$, but not the termination one. This is because the user must provide a projection relation from the $\mathbb{Z}$ quotient type back into the constructive type $\mathbb{N}$.

Even if we could avoid these 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. 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 extending our 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

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

```
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[3] 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 be-

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

cause 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)›
```

The proof obligations for pattern compatibility and completeness are next. 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.

Various theorems are made available, such as:

- Case analysis $(\bigwedge$n. ?x = n $\Longrightarrow$?P$) \Longrightarrow$?P
- Elimination rules (partial) ⟦vdm-factorial ?x = ?y; vdm-factorial-dom ?x; $\bigwedge$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⟧ $\Longrightarrow$?P⟧ $\Longrightarrow$?P
- Induction rules (partial) ⟦vdm-factorial-dom ?a0.0; $\bigwedge$n. ⟦vdm-factorial-dom n; ⟦pre-vdm-factorial n; n ≠ 0⟧ $\Longrightarrow$?P (n − 1)⟧ $\Longrightarrow$?P n⟧ $\Longrightarrow$?P ?a0.0
- Simplification rules (partial) vdm-factorial-dom ?n $\Longrightarrow$ vdm-factorial ?n = (if pre-vdm-factorial ?n then if ?n = 0 then 1 else ?n ∗ vdm-factorial (?n − 1) else undefined)

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 gen-VDMNat-term and int-ge-less-than.

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.

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)

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
- Induction rules (total) (⋀n. (⟦pre-vdm-factorial n; n ≠ 0⟧ ⟹ ?P (n − 1)) ⟹ ?P n) ⟹ ?P ?a0.0
- 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)

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›

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′

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

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 gen-set-term and finite-psubset

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

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)›

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.

Note we omit such lemma over termination and precondition for the VDM-Nat 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›

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›

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-set choice is part of the filtering predicate.

Following the general translation strategy for maps [?], we define the function precondition using inv-Map 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)›
```

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. 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›
```

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›
```

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 ≠ {}›
```

5.4 VDM recursion involving complex measures

The class of recursive examples shown so far have cover a wide range of situations, and have a good level of automation. Nevertheless, the same strategy can also be applied for me complex recursive definitions. The cost for the VDM user is the need of a more involved @IsaMeasure definition and the highly likely need for extra user-defined lemmas. These lemmas can be defined in VDM itself using the @Lemma annotation, which will be translated to Isabelle as any other boolean expression would.

To illustrate this, we define in VDM the (in)famous Ackermann function[4], which is a staple example of complex recursion, as

```
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;
```

Note that the VDM measure is not defined, and that the @IsaMeasure uses a construct from Isabelle called pair-less. It is part of Isabelle's machinery of concrete orders for SCNP problems [?]. It considers recursions over multiple parameters, where some might increase the number of calls (e.g. size-change). We are not aware of a mechanism to define such measures in VDM.

We instantiate pair-less to VDMNat. It is defined as the lexicographic product over the transitive closure of a totally ordered relation between VDMNat inputs[5].

If VDM measures were over relations, the Ackermann measure could be defined defined in VDM, assuming a standard definition of transitive closure[6], as

```
pair_less_VDMNat: () -> set of ((nat * nat) * (nat * nat))
pair_less_VDMNat() == lex_prod[nat, nat](less_than_VDMNat(), less_than_VDMNat());

less_than_VDMNat: () -> set of (nat * nat)
less_than_VDMNat() == transitive_closure[nat]({ mk_(z', z) | z', z : nat & z' < z });

lex_prod[@A,@B]: set of (@A * @A) * set of (@B * @B) -> set of ((@A * @B) * (@A * @B))
lex_prod(ra,rb) ==
    { mk_(mk_(a, b), mk_(a', b')) | a, a': @A, b, b': @B &
        mk_(a,a') in set ra or a = a' and mk_(b, b') in set rb };
```

That is, the lexicographic product of possibilities that are ordered in its

[4] https://en.wikipedia.org/wiki/Ackermann_function

[5] Details of this definition in the VDMToolkit.thy file within the distribution in https://github.com/leouk/VDM_Toolkit.

[6] The vdmlib/Relations.vdmsl provides such definition in the VDM toolkit distribution.

parameters. The translation process is the same as above and produces the Isabelle below.

```
definition pre-ack :: ‹VDMNat ⇒ VDMNat ⇒ 𝔹› where
  ‹pre-ack m n ≡ inv-VDMNat m ∧ inv-VDMNat n›

lemmas pre-ack-defs = pre-ack-def inv-VDMNat-def

function (domintros) ack :: ‹VDMNat ⇒ VDMNat ⇒ VDMNat› where
  ‹ack m n = (if pre-ack m n then
                if m = 0 then n+1
           else if n = 0 then ack (m−1) 1
           else               ack (m−1) (ack m (n−1))
           else               undefined)›

abbreviation ack-wf :: ‹((VDMNat × VDMNat) × (VDMNat × VDMNat)) VDM-
Set› where
  ‹ack-wf ≡ pair-less-VDMNat›

termination
  apply (relation ack-wf)
  using wf-pair-less-VDMNat apply blast
    apply (simp add: inv-VDMNat-def l-pair-less-VDMNat-I1 pre-ack-def)
   apply (simp add: inv-VDMNat-def pre-ack-def)
  by (simp add: inv-VDMNat-def pair-less-VDMNat-def pre-ack-def)
```

We also show that this version of Ackermann with VDMNat is equivalent to the usual Isabelle definition using ℕ. We omit details here, but have proved that they are equivalent by induction

```
theorem ack-correct: ‹ack′ m n = ack m n›
  apply (induction ‹m› ‹n› rule: ack′.induct) by (simp add: pre-ack-defs)+
```

In general, each complex recursion function will require such a setup. Fortunately, Isabelle has a number of options available. Yet, in general, the more complex the recursion, the more users will have to provide further machinery. For example, the next two examples for Nipkow's permutation function [?] and Takeuchi's function[7] require a rather ellaborate setup (see Appendix 7).

6 Discussion and conclusion

In this paper we present a translation strategy from VDM to Isabelle for recursive function over basic types, sets and maps. We also present how the strategy works for more complex recursion, such as the Ackermann's function.

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

The full VDM and Isabelle sources and proofs can be found at the VDM toolkit repository at ./plugins/vdm2isa/src/main/resources/RecursiveVDM.*[8].

Future work. We are implementing the translation strategy in the vdm2isa plugin, which should be available for the VDM-VSCode [?] extension in the next release.

Acknowledgements. We appreciated discussions with Stephan Merz on pointers for complex well-founded recursion proofs in Isabelle, and with Nick Battle on limits for VDM recursive measures.

7 Further complex examples

```
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;
```

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

7.1 Nipkow's permutation function

```
definition
  pre-perm :: ‹VDMInt ⇒ VDMInt ⇒ VDMInt ⇒ 𝔹›
  where
  ‹pre-perm m n r ≡
     inv-VDMInt m ∧ inv-VDMInt n ∧ inv-VDMInt r ∧
     ((0 < r ∨ 0 < n) ⟶ m+n+r > 0)›

lemmas pre-perm-defs = pre-perm-def inv-VDMInt-def inv-True-def

lemma l-pre-perm-trivial[simp]: (pre-perm m n r) = ((0 < r ∨ 0 < n) ⟶ m+n+r
> 0)
  unfolding pre-perm-def inv-VDMInt-def by simp

function (domintros)
  perm :: ‹VDMInt ⇒ VDMInt ⇒ VDMInt ⇒ VDMInt›
  where
  ‹perm m n r =
```

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

```
    (if pre-perm m n r then
      (     if 0 < r then perm m (r−1) n
       else if 0 < n then perm r (n−1) m
       else m)
    else
      undefined)
  ⟩
  by (pat-completeness, auto)

definition
   perm-wf-rel :: ⟨((VDMInt × VDMInt × VDMInt) × (VDMInt × VDMInt ×
VDMInt)) VDMSet⟩
  where
  ⟨perm-wf-rel ≡
      ({ ((m, r−1, n), (m, n, r)) | m r n . 0 < r ∧ pre-perm m n r}
      ∪
      { ((r, n−1, m), (m, n, r)) | m r n . ¬ 0 < r ∧ 0 < n ∧ pre-perm m n r }
      )
  ⟩

lemma l-perm-wf-rel: wf perm-wf-rel
proof −
```

The Isabelle measure projection reflects the VDM measure: the sum of each parameter. ending up at zero (i.e. if negative, ignore).

```
  have perm-wf-rel ⊆ measure (λ (m, r, n) . nat (max 0 (m+r+n)))
    apply (intro subsetI, case-tac x)
    apply (simp add: perm-wf-rel-def case-prod-beta max-def)
    apply (elim disjE conjE, simp)
     apply (intro impI conjI, simp-all)
```

The setup with perm-wf-rel works here if the pre-perm specifically curbs negative sums of parameters. Yet, the termination proof fails. Tried various variations on gr or geq etc, no luck. Trying the (<∗mlex∗>) style nest instead.

```
    nitpick
    done
  then show ?thesis
    by (rule wf-subset [OF wf-measure])
qed
```

With the added precondition on pre-perm about case when sum has to be positive, and the well founded proof above, the result worked well

```
termination
  apply (relation ⟨perm-wf-rel⟩)
    apply (simp add: l-perm-wf-rel)
   apply (simp-all add: perm-wf-rel-def)
  done
```

7.2 Takeuchi's function (from HOL/Examples/Functions.thy)

```
function
  tak :: VDMInt ⇒ VDMInt ⇒ VDMInt ⇒ VDMInt
  where
  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))
  by auto

lemma tak-pcorrect:
  tak-dom (x, y, z) ⟹ tak x y z = (if x ≤ y then y else if y ≤ z then z else x)
  thm tak.pinduct tak.psimps
  apply (induction x y z rule: tak.pinduct)
  by (simp add: tak.psimps)

definition tak-m1 where tak-m1 = (λ(x,y,z). if x ≤ y then 0 else 1)
definition tak-m2 where tak-m2 = (λ(x,y,z). nat (Max {x, y, z} − Min {x, y, z}))
definition tak-m3 where tak-m3 = (λ(x,y,z). nat (x − Min {x, y, z}))

lemma (((x1,y1,z1),(x2,y2,z2)) ∈ tak-m3 <∗mlex∗> {}) ⟷ (nat (x1 − Min {x1,
y1, z1}) < nat (x2 − Min {x2, y2, z2}))
  apply (simp add: mlex-iff)
  unfolding tak-m3-def

  by simp

lemma ‹((x1,y1,z1),(x2,y2,z2)) ∈ (tak-m2 <∗mlex∗> (tak-m3 <∗mlex∗> {}))›
  apply (simp add: mlex-iff)
  unfolding tak-m3-def tak-m2-def
  apply (simp only: case-prod-beta fst-conv snd-conv, clarify)
  oops

lemma l-call1:
  ‹x > y ⟹ ((x−1,y,z),(x,y,z)) ∈ tak-m1 <∗mlex∗> (tak-m2 <∗mlex∗> (tak-m3
<∗mlex∗> {}))›
  apply (simp add: mlex-iff)
  unfolding tak-m3-def tak-m2-def tak-m1-def
  apply (simp add: case-prod-beta)
  unfolding min-def max-def
  by (simp-all split:if-splits)

lemma l-call2:
  ‹x > y ⟹ ((y−1,z,x),(x,y,z)) ∈ tak-m1 <∗mlex∗> (tak-m2 <∗mlex∗> (tak-m3
<∗mlex∗> {}))›
  apply (simp add: mlex-iff)
  unfolding tak-m3-def tak-m2-def tak-m1-def
  apply (simp add: case-prod-beta)
  unfolding  min-def max-def
  by simp
```

```
lemma l-call3:
  ‹x > y ⟹ ((z−1,x,y),(x,y,z)) ∈ tak-m1 <∗mlex∗> (tak-m2 <∗mlex∗> (tak-m3
<∗mlex∗> {}))›
  apply (simp add: mlex-iff)
  unfolding tak-m3-def tak-m2-def tak-m1-def
  apply (simp add: case-prod-beta)
  unfolding  min-def max-def
  by simp

lemma l-call4:
  ‹x >y ⟹  tak-dom (z − 1, x, y) ⟹
       tak-dom (y − 1, z, x) ⟹
       tak-dom (x − 1, y, z) ⟹
       ((tak (x − 1) y z, tak (y − 1) z x, tak (z − 1) x y), x, y, z)
       ∈ tak-m1 <∗mlex∗> tak-m2 <∗mlex∗> tak-m3 <∗mlex∗> {}›
  apply (simp add: mlex-iff)
  apply (simp add: tak-pcorrect)
  unfolding tak-m3-def tak-m2-def tak-m1-def
  apply (simp add: case-prod-beta)
  unfolding  min-def max-def
  by simp

termination
  apply (relation tak-m1 <∗mlex∗> tak-m2 <∗mlex∗> tak-m3 <∗mlex∗> {})
      apply (simp add: wf-mlex)
     apply (simp add: l-call1)
     apply (simp add: l-call2)
     apply (simp add: l-call3)
  by (simp add: l-call4)

theorem tak-correct: tak x y z = (if x ≤ y then y else if y ≤ z then z else x)
  by (induction x y z rule: tak.induct) auto
```