VDM recursive functions in Isabelle/HOL

Leo Freitas[1] and Peter Gorm Larsen[2]

[1] School of Computing, Newcastle University,
leo.freitas@newcastle.ac.uk
[2] DIGIT, Aarhus University, Department of Engineering,
pgl@ece.au.dk

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, and a translation and proof environment for VDM in Isabelle/HOL.

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

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 the VDM to Isabelle/HOL translation strategy and implementation as a plugin to VDMJ [1] and extension to VDM-VSCode [8].

Isabelle uses literate programming, where formal specification, proofs and documentation are all within the same environment. We omit proof 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

Our VDM to Isabelle/HOL translator caters for a wide range of the VDM-SL 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 [4,8]. Although not all VDM patterns are allowed, the translator copes with most, and where it does not, there is a corresponding equivalent.

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 terminates. It then generates proof obligations to ensure totality and termination.

Finally, our translation strategy follows the size-change termination (SCT) proof strategy described in [7,3]. In particular, its SCP (polynomial) and SCNP (non-polynomial) subclasses of recursive definitions within the SCT, which permits efficient termination 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.

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. 2 = 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. As 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 might be 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's 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

An important aspect of every recursive definition is an argument that justifies its termination. Otherwise, the recursion might go into 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 a VDM recursive definition is one for calculating the factorial of a given natural number

```
fact: nat -> nat
fact(n) == if n = 0 then 1 else n * fact(n - 1)   measure n;
```

The fact recursive measure uses the n input itself as its result. 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. They are trivial to discharge in Isabelle given the measure definition expanded is just $\forall$ n. n $\neq$ 0 $\longrightarrow$ 0 $\leq$ n - 1 and $\forall$ n. n $\neq$ 0 $\longrightarrow$ n - 1 $<$ n.

```
Proof Obligation 1: (Unproved) fact; measure_fact: total function obligation
(forall n:nat & is_(measure_fact(n), nat))

Proof Obligation 2: (Unproved) fact: subtype obligation
(forall n:nat & (not (n = 0) => (n - 1) >= 0))

Proof Obligation 3: (Unproved) fact: recursive function obligation
(forall n:nat & (not (n=0) => measure_fact(n) > measure_fact((n-1))))
```

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 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 5.4).

In Isabelle, recursive definitions can be provided through primitive recursion over inputs that are constructively defined, or more general function definitions that produce proof obligations. The former insists on definition for each type constructor and only provides simplification rules; whereas the latter allows 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 [5].

Isabelle recursive functions require a proof obligation that its parameters represent a constructive and compatible pattern, and that recursive calls terminate. Constructive patterns covers all constructors in the data type being used in the recursion inputs (i.e., one equation for each of the constructors of $\mathbb{N}$, hence one involving 0 and another involving Suc n). Compatible patterns cover the multi-

ple 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 recursion.

The termination relation must be well-founded, which means have a well-ordered induction principle over a partially ordered relation defined as[2]

```
wf (r::('a × 'a) set) =
(∀P::'a ⇒ 𝔹. (∀x::'a. (∀y::'a. (y, x) ∈ r ⟶ P y) ⟶ P x) ⟶ (∀x::'a. P x))
```

A definition that Isabelle discovers all three proofs is

```
fun fact' :: ‹ℕ ⇒ ℕ› where ‹fact' n = (if n = 0 then 1 else n * (fact' (n - 1)))›
```

This definition is quite similar in VDM. Nevertheless, VDM basic types widening rules necessitate we translate them to VDMNat. The same version of fact defined for ℤ will fail with the error that "Could not find lexicographic termination order". That is, Isabelle manages to discharge the pattern proofs, but not the termination one. This is because the user must provide a projection relation from the ℤ quotient type back into the constructive type ℕ.

Even if we could avoid this translation technicality, the same problem would occur for recursion over non-constructive types, such as sets or maps. They require recursion over finite sets, which are defined inductively. The only easy recursive translations are those involving lists, given lists in Isabelle are defined constructively and VDM sequences map directly.

Therefore, defining recursive functions over non-constructive types entail compatibility and completeness proofs. They also 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.

[2] Details on well-ordered induction are in Isabelle's Wellfounded.thy theory.

5 VDM recursion translation strategy

We want to identify a translation strategy that will cater for issues described above not only for basic types, but also for sets, sequences, maps, as well as mutual recursion.

The VDM-SL AST tags all recursive functions, even those without an explicit measure. All such functions will be translated using Isabelle's fun syntax, which will attempt to discover proofs for compatibility and termination. For our setup of the fact example, where Isabelle discovers the termination proof.

If that had failed, the user could define a VDM @IsaMeasure annotation. VDM annotations are comments that will be processed according to specific implementations [2,8]. If the user does not provide an @IsaMeasure annotation and fun fails, then it is up to the user to figure out the necessary proof setup.

The @IsaMeasure annotation defines a well-founded measure relation that will participate in the setup for Isabelle termination proof. For example, for the fact example, the user would have to write an annotation before the VDM measure.

```
--@IsaMeasure( { (n -1, n) | n : nat & n <> 0 } )
```

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

During translation, the vdm2isa 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 a 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, then the plugin will try to automatically infer what the measure relation should be based on the structure of the recursive function definition. If this fails, then the user is informed. Still, even if measure-relation synthesis succeeds, the user still has 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 of the overall translation strategy, see examples in the distribution[3] and [4]. The translation strategy imposes various implicit VDM checks as explicit predicates. For example, VDM sets are always finite, and type invariants over set, sequence and map elements must hold for every element.

5.1 Recursion over VDM basic types (nat, int)

Following the general translation strategy [4], we first encode the implicit precondition, which insists that the given parameter n is a VDMNat, alongside a list of defining constants that are useful for proof strategy synthesis.

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

```
definition pre_fact :: ‹VDMNat ⇒ 𝔹› where ‹pre_fact n ≡ inv_VDMNat n›
lemmas pre_fact_defs = pre_fact_def
```

Next, we define factorial recursively. When the precondition fails, we return undefined, which is a term that cannot be reasoned with in Isabelle The domintros tag tells Isabelle to generate domain predicates, in case this function is not total. Domain predicates are important to our strategy because every VDM function will be undefined, when applied outside its precondition. It also generates domain-predicate sensitive proof rules listed below.

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

The proof obligations for pattern compatibility and completeness are discharged with the usual Isabelle proof strategy for simple recursive patterns with the pat_completeness method. In the general case discussed in Section 5.4, the user might have goals to discharge. Isabelle proves various theorems abou case analysis and (partial) rules for elimination, induction and simplification.

Partial rules require the domain predicate fact_dom as an assumption. It represents a well-founded relation that ensures termination. That is, if the user does not want (or knows how) to prove termination, such domain predicates will follow every application of definitions, hence documenting 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 automate.

Termination proof is discharged by establishing a well-founded relation associated with the function recursive call(s) with respect to its declaration. The @IsaMeasure annotation is translated as an Isabelle abbreviation. 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 from the if-statement. More complex definitions will have more involved filters. We use abbreviation instead of definition to avoid needing to expand the defined term in proofs.

```
abbreviation fact_wf :: ‹(VDMNat × VDMNat) set› where
  ‹fact_wf ≡ { (n - 1, n) | n . pre_fact n ∧ n ≠ 0 }›
```

Given fact definition is simple (non-mutual, single call-site, easy measure relation choice) recursion, its setup is easy to establish well-foundedness. For recursions of this nature, we can piggyback on Isabelle machinery to prove well foundedness by using 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-founded relation fact_wf. Isabelle has proofs about the well foundedness of int_ge_less_than. Thus, making the proof our term being well-founded trivial for sledgehammer. As part of the translation strategy, we define (and attempt to

discover the proof of) the following lemma. This follows the strategy described in [6].

```
lemma l_fact_term_wf: ‹wf (gen_VDMNat_term fact_wf)›
```

Finally, we prove termination using the previously proved lemma using the relation. This simplifies the goal into well-foundedness of termination relation and that the precondition implies it, both of which are easily proved with simplification in this case.

For this example, subgoals are proved with sledgehammer. In general, the user will be have to either find the proof, or deal with domain predicates involving the recursive call. After the termination proof is discharged, Isabelle provides versions of rules for elimination, induction and simplification that are total and do not depend on the domain predicates.

To make sure our choice does not lead to the empty relation, we ensure that 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_fact_term_valid: ‹(gen_VDMNat_term fact_wf) = fact_wf›
```

5.2 Recursion over VDM sets

Next, we extend the translation strategy for VDM sets. For this, we use a function that sums the elements of a set.

```
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;
```

Most common VDM recursion over sets, consume the set by picking an arbitrary set element and then recurse 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. VDM measures are not suitable for Isabelle proofs, given Isabelle requires a relation; hence, VDM measures are mostly ignored. They might still be useful during proofs as potential witnesses to existentially-quantified goals.

The implicit VDM checks are defined as the precondition, which ensures that the given set only contains 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
```

We define the VDM recursive function in Isabelle next. It checks whether the given set satisfy the function precondition, returning undefined if not. Each case is encoded pretty much 1-1 from VDM after that. 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 trivial. The measure relation defined with @IsaMeasure is next. It is defined as the smaller set after picking e 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 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 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 in the relation pair is well-founded. This makes the proof of well-foundedness easy for sledgehammer.

```
lemma l_sumset_rel_wf: ‹wf (gen_set_term sumset_wf_rel)›
```

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

Fortunately, for most simple situations, this is easy to decompose in general. The translation strategy takes the @IsaMeasure expression and decomposes its parts, such that the filtering predicates are assumptions, and the element in the relation belongs to the well-founded measure chosen. This is defined in the next lemma, which require some manual intervention until 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-foundedness under the function precondition and the filtering case ($s \neq \emptyset$) 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 is that the set is finite. With this, we try the termination proof and sledgehammer find proofs for all subgoals.

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, where sledgehammer find the proof once more.

```
lemma l_pre_fact_wf_rel:
  ‹⟦pre_fact n; n ≠ 0⟧ ⟹ (n - 1, n) ∈ gen_VDMNat_term fact_wf›
```

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 [8]: it provides a concrete example for the function input parameters. This witness is useful for existentially quantified predicates of involved termination proofs (see Section 5.4).

5.3 Recursion over VDM maps

Recursive functions over maps are a special case of sets, given map recursion usually iterates over the map's domain. For example, the function that sums the elements of the map's range is 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 recursing on the map filtered by removing the chosen element, until the map is empty and the function terminates. 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 [4], 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, then Isabelle require the proof such set is finite. Given both domain and range sets are finite, this should be easy, 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
```

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, then the map results in a non nil value; whereas, if the element does not belong to the domain, then the map results in a nil value. This makes all maps total, where

values outside the domain map to nil. The Isabelle translation and compatibility proof follows patterns used before.

```
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-founded relation is translated next, where the precondition is included as part of the relation's filter. The {d} -⩤ m corresponds to the VDM domain anti-filtering operator {d}<-:m.

```
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-founded lemma over the recursive measure relation, there is no available Isabelle help, and projecting the domain element of the maps within the relation is awkward. Thus, we have to prove the well-founded lemma and 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 foundedness of the finite_psubset. It uses the provided VDM measure expression to extract the right projection of interest, then follows the proof for finite_psubset, where sledgehammer can find the final steps.

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

```
lemma l_sum_elems_wf: ‹wf sum_elems_wf›

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›

lemma l_sum_elems_wf_valid: ‹sum_elems_wf ≠ {}›
```

Finally, we also prove that the well founded termination relation 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.

5.4 VDM recursion involving complex measures

The class of recursive examples shown so far have covered a wide range of situations, and have a good level of automation. Nevertheless, the same strategy can also be applied for more complex recursive definitions. The cost for the VDM user is the need of involved @IsaMeasure definitions and the highly likely need for extra user-defined lemmas. These lemmas can be defined in VDM itself using the @Lemma annotation. To illustrate this, we define in VDM the (in)famous Ackermann function[4], which is a staple example of complex recursion.

```
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 [6]. 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 as the lexicographic product over the transitive closure of a totally ordered relation between inputs[5]. If VDM measures were over relations, then the Ackermann measure could be defined in VDM, assuming a standard definition of transitive closure[6].

```
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() == trans_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 represents the lexicographic product of possibilities that are ordered in its parameters. Translation then outputs:

```
definition pre_ack :: ‹VDMNat ⇒ VDMNat ⇒ 𝔹› where
  ‹pre_ack m n ≡ inv_VDMNat m ∧ inv_VDMNat n›
lemmas pre_ack_defs = pre_ack_def

function (domintros) ack :: ‹VDMNat ⇒ VDMNat ⇒ VDMNat› where
  ‹ack m n = (if pre_ack m n then
                if m = 0 then n+1
```

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

[5] Details of this definition are in the VDMToolkit.thy within the distribution.

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

```
                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)) VDMSet›

  where ‹ack_wf ≡ pair_less_VDMNat›
```

Compatibility and termination proofs are similar, despite the more complex measure, because of available Isabelle automation. 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 recursive 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 automation.

5.5 Harder examples

We can handle all examples from [7]. We show here some that require an elaborate setup. For example, the permutation function shows permuting decreasing parameters with an involved measure. The precondition was required for finishing the termination proof and shows an example why proofs over ℤ can be harder. This illustrated how Isabelle (failed) proofs helped to improve the VDM specification.

```
perm: int * int * int -> int
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
--@IsaMeasure({mk_(mk_(m, r-1, n), mk_(m,n,r)) | ... & 0 < r} union
--            {mk_(mk_(r, n-1, m), mk_(m,n,r)) | ... & not 0 < r and 0 < n})
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;
```

```
definition pre_perm :: ‹VDMInt ⇒ VDMInt ⇒ VDMInt ⇒ 𝔹› where
  ‹pre_perm m n r ≡ ((0 < r ∨ 0 < n) ⟶ m+n+r > 0)›
lemmas pre_perm_defs = pre_perm_def inv_VDMInt_def inv_True_def

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

definition ‹perm_wf_rel ≡
  { ((m, r-1, n), (m, n, r)) | m r n . pre_perm m n r ∧ 0 < r } ∪
  { ((r, n-1, m), (m, n, r)) | m r n . pre_perm m n r ∧ ¬ 0 < r ∧ 0 < n }›
```

Its measure relation contains elements for each recursive call, filtered for the corresponding if-then case. The proof of well-foundedness of such measure relations involving multiple recursive calls require further proof engineering, which is stated in the next lemma.

```
lemma l_perm_wf_rel_VDM_measure:
  ‹perm_wf_rel ⊆ measure (λ (m, r, n) . nat (max 0 (m+r+n)))›

lemma l_perm_wf_rel: ‹wf perm_wf_rel›
```

The Isabelle measure definition projects the less-than ordered inverse image[7] of a given function as the recursive measure relation. Here the VDM-defined measure is given as such measure function projection. This highlights to the VDM user the relationship (and differences) between VDM and Isabelle recursive measures.

The setup works here if the pre_perm specifically curbs negative sums of parameters. This was not immediately obvious. With the added precondition the termination proof is discovered with sledgehammer.

Finally, the Takeuchi's function[8], which contains both permutation and inner recursion is defined next, where the important part is the SCNP setup using multi-sets [6]. That is needed because ordered lexicographic products are not strong enough to capture this type of recursion. It has not implicit precondition. The translation strategy works for these definitions, yet stands little chance of finding proofs automatically.

```
function (domintros) 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))›
```

Next is an example of how one has to keep domain predicates as assumptions prior to termination proofs.

```
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)›
```

[7] That is, measure = inv_image less_than. Inverse image is defined as inv_image r f = {(x, y) | (f x, f y) ∈ r}.

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

Each case (including the non-recursive call) is represented in the SCNP setup, and then their measure-lexicographic[9] composition is used as the measure relation for the termination proof.

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}))›

The termination proof uses the measure relation tak_m1 <*mlex*> tak_m2 <*mlex*> tak_m3 <*mlex*> ∅. It requires user-defined lemmas for each of the four cases.

With the total version of induction and simplification rules are available. Then, it is possible to prove its rather simpler equivalence.

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

5.6 Mutual recursion

Finally, we handle mutual recursion. VDM has few bounds on mutually recursive definitions. We reuse the VDM type checker cyclic-measure discovery algorithm in order to deduce related recursive calls. Note this is not complete and may fail to discover involved recursive calls, such as those under some lambda-term result, for example. If that recursive call graph discovery fails, then users have to annotate at least one of the mutually recursive definitions with an @IsaMutualRec annotation listing all expected function names involved.

Mutually recursive definitions proof obligations refer to each others measure functions. Isabelle requires all related functions to be given in a single definition, where Isabelle's sum (or union) types are used in the proof setup. Where needed, termination proof setup will be involved and with limited automation. Here is a simple example and its translation, which works with fun syntax, hence has no further proof obligation needs. Note that in this case using fun syntax, we declare implicit preconditions as simplification rules.

```
--@IsaMutualRec({odd})
even: nat -> bool
even(n) == if n = 0 then true else odd(n-1) measure n;

--@IsaMutualRec({even})
odd: nat -> bool
odd(n) == if n = 0 then false else even(n-1) measure n;
```

definition pre_even :: ‹VDMNat ⇒ 𝔹› where [simp]: ‹pre_even n ≡ inv_VDMNat n›

[9] The measure-lexicographic product f <*mlex*> R is represented as the inverse image of the lexicographic product f <*mlex*> R = inv_image (less_than <*lex*> R) (λx. (f x, x)).

```
definition pre_odd  :: ‹VDMNat ⇒ 𝔹› where [simp]: ‹pre_odd n ≡ inv_VDMNat n›
fun (domintros) even :: ‹VDMNat ⇒ 𝔹› and odd  :: ‹VDMNat ⇒ 𝔹› where
  ‹even n = (if pre_even n then (if n = 0 then True else odd (n-1)) else undefined)›
| ‹odd  n = (if pre_odd n then (if n = 0 then False else even (n-1)) else undefined)›
```

6 Discussion and conclusion

In this paper we present a translation strategy from VDM to Isabelle for recursive functions over basic types, sets and maps, as well as mutual recursion. We present how the strategy works for complex recursion.

The complex recursion examples hint at possible VDM recursive measure extensions to use a combination of measure relations and functions. The full VDM and Isabelle sources and proofs can be found at the VDM toolkit repository at RecursiveVDM*.thy[10].

Future work. We are implementing the translation strategy in the vdm2isa plugin, which should be available soon.

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.

References

1. Battle, N.: VDMJ User Guide. Tech. rep., Fujitsu Services Ltd., UK (2009)
2. Battle, N.: VDMJ Annotaions Guide (2022)
3. Ben-Amram, A.M., Codish, M.: A sat-based approach to size change termination with global ranking functions. In: TACAS. pp. 218–232. Springer (2008)
4. Freitas, L.: A VDM translation strategy to isabelle (Nill Full game) (2016)
5. Krauss, A.: Defining Recursive Functions in Isabelle/HOL. Technical University of Munich (2021)
6. Krauss, A., Heller, A.: A mechanized proof reconstruction for scnp termination. Tech. rep., Technical University of Munich (2012)
7. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: POPL. pp. 81–92. POPL '01, ACM, New York USA (2001)
8. Rask, J.K., et al.: Advanced VDM Support in Visual Studio Code. In: 20th International Overture Workshop. pp. 35–50 (2022)

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