resources

Leo Freitas

November 2, 2022

Contents

```
theory RecursiveSet
  imports VDMToolkit
begin
```

1 Recursive partial recursive functions

In Isabelle, recursive functions must discharge proof obligations on:

1. pattern completeness:

   This relates to all patterns in a constructive type being refered to (e.g., 0::′a and Suc n for ℕ)).

2. pattern compatibility:

   This relates to multiple way patterns can be constructed that boils down to the pattern completeness cases (e.g., n + (2::′a) being simply multiple successor calls over constructors Suc (Suc 0)).

That is important to ensure that recursion is well structured (i.e., recursive calls will not get stuck because call constructs are not available). For example, if you miss the 0::′a case, eventually the Suc n case will reach zero and fail.

A final proof obligation is on termination: the recursion is well-founded. This has to be proved whenever properties of defined function are meant to be total.

For example, a function that finds the zero of functions can be given as:

```
function (domintros) findzero :: (nat ⇒ nat) ⇒ nat ⇒ nat
where
findzero f n = (if f n = 0 then n else findzero f (Suc n))
```

```
  by pat-completeness auto

print-theorems

term findzero-dom
term findzero-rel
thm findzero.domintros impI
    findzero-rel.intros
find-theorems name:Wellfounded.acc
```

Various theorems are made available, such as:

$(\bigwedge$f n. ?x = (f, n) $\Longrightarrow$?P) $\Longrightarrow$?P[display] Cases analysis

$[\![$findzero ?x ?xa = ?y; findzero-dom (?x, ?xa); $\bigwedge$f n. $[\![$?x = f; ?xa = n; ?y = (if f n = 0 then n else findzero f (Suc n)); findzero-dom (f, n)$]\!]$ $\Longrightarrow$?P$]\!]$ $\Longrightarrow$?P[display] Elimination rules

$[\![$findzero-dom (?a0.0, ?a1.0); $\bigwedge$f n. $[\![$findzero-dom (f, n); f n $\neq$ 0 $\Longrightarrow$?P f (Suc n)$]\!]$ $\Longrightarrow$?P f n$]\!]$ $\Longrightarrow$?P ?a0.0 ?a1.0[display] Induction rules

findzero-dom (?f, ?n) $\Longrightarrow$ findzero ?f ?n = (if ?f ?n = 0 then ?n else findzero ?f (Suc ?n))[display] Simplificaiton rules

Note the last two are partial, module a domain predicate findzero-dom, which represents a well-founded relation that ensures termination. These p-rules can be simplified into total rules that do not depend on a domain predicate, which can compicate proofs.

2 Exampe of recursive functions with non-constructive types

Recursing on non-constructive types (e.g., sets, integers, etc.) entail more involved compatibility and completeness proofs. They also usually lead to partial function definitions, given Isabelle can't tell whether termination is immediatelly obvious.

In VDM, however, recursive functions on sets (as well as map domains) are common.

In our vdm2isa translator, we impose various implicit VDM checks as explicit predicates in Isabelle. In VDM, sets are always finite, and structural invariants are declared for types.

Our example recursive function is given a set of $\mathbb{N}$ and return their sum. In VDM, because of various type widening rules (e.g., 0 − x returns an integer result, whereas in Isabelle this remains a $\mathbb{N}$.). We encode VDM corresponding type as VDMNat. This is represented in Isabelle as $\mathbb{Z}$ in order to allow for VDM type widening rules during translation.

The function is defined in VDM as:

```
        sumset: set of nat -> nat
        sumset(s) == if s = {} then 0 else let e in set s in sumset(s - {e}) + e;
```

It consumes the set by picking each set element and summing them to the recursive call until the set is empty.

In Isabelle, the implicit VDM checks are defined as the precondition, which ensures that the given set contains only natural numbers, and is finite.

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

Termination proof is achieved by establishing a well-founded relation associated with the function recursive call with respect to its declaration.

In our case, that is the smaller set after picking e (s − {SOME e. e ∈ s}) and the set used at definition, leading to the pairs (s − {SOME e. e ∈ s}, s). We ensure all the s involved are not empty and satisfy the function precondition (pre-sumset).

Given this is a simple (non-mutual, single call-site, easy set element choice) recursion, thankfully the setup is not as complex to establish well-foundedness. We piggyback on some Isabelle machinery by using the term:

finite-psubset[display]

It establishes that a relation where the first element is strictly smaller set than the second element in the relation pair. This makes the proof of well-foundedness easy for sledgehammer, which is important in order for translated code be easier to prove.

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

definition
  sumset-term ::(VDMNat VDMSet × VDMNat VDMSet) set where
  sumset-term ≡ finite-psubset ∩ sumset-wf-rel
```

Termination requires well-founded relation, so we prove that function sumset termination relation is well-founded using sledgehammer.

```
lemma l-sumset-term-wf: wf sumset-term
  by (simp add: sumset-term-def wf-Int1)
```

Moreover, once we establish well-foundedness, we need to get to the termination relation from the filtering predicate defined through the precondition (i.e. the precondition helps establish the terminating relation).

In this case, the only needed term for Isabelle to establish termination is set finiteness, however, we insist on the whole precondition to ensure that the intended VDM meaning is maintained.

```
lemma l-pre-sumset-sumset-term:
  pre-sumset s ⟹ s ≠ {} ⟹ x = (SOME x. x ∈ s) ⟹ (s − {x}, s) ∈ sumset-term
  apply (simp add: pre-sumset-def sumset-term-def)
  by (metis Diff-subset l-invVDMSet-finite-f member-remove psubsetI remove-def
some-in-eq)
```

Finally, we can define our recursive function in Isabelle. It checks whether the given set satisfy the function precondition. If it doesn't, undefined is returned. If it does, then each case is encoded pretty much 1-1 from VDM using Hilbert's choice operator.

```
function (domintros)
  sumset :: VDMNat VDMSet ⇒ VDMNat
  where
  sumset s =
    (if pre-sumset s then
        (if s = {} then
            0
         else
            let e = (SOME 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 tactic. For more general examples, if that fails, sledgehammer should be used.

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

Next, we have to discharge the termination proof, which is given as All sumset-dom 1. All sumset-dom[display]

```
  thm termination
  apply (rule termination[of sumset-term])
```

We follow the strategy of using the termination relation and well formedness, which transforms the mysterious/abstract domain predicate into two new subgoals All sumset-dom 1. wf sumset-term 2. $\bigwedge$s x. ⟦pre-sumset s; s ≠ ∅; x = (SOME x. x ∈ s)⟧ ⟹ (s − {x}, s) ∈ sumset-term[display]

The first goal is direclty discharged with wf sumset-term.

```
   apply (simp add: l-sumset-term-wf)
```

Finally, we show that termination relation is entailed by function precondition.

```
  by (simp add: l-pre-sumset-sumset-term)
```

Is the sumset termination relaiton non-trivial? That is, we have some solutions within the finite subsets representing the recursive wellfounded relation.

```
lemma l-sumset-term-not-empty: sumset-term ≠ {}
  apply safe
  find-theorems elim
  apply (erule equalityE)
  find-theorems {} ⊆ -
  find-theorems - ⊆ {} elim
  find-theorems (- ⊆ -) = -
  thm subset-iff-psubset-eq subset-eq
  apply (simp add: subset-eq)
  unfolding sumset-term-def
  apply simp
  apply (erule-tac x={1} in allE)
  by (auto simp add: pre-sumset-def inv-VDMNat-def)
```

This is suboptimal. Let's then show it's the relation (i.e. finite subset trick to make wellfounded induction proof easier, does not compromise the wellfounded relation itself).

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


end
theory RecursiveVDMNat
  imports VDMToolkit
begin
```

VDM expressions with basic-typed (nat, int) variables (e.g. x − y) have specific type widening rules (e.g. if both variables are nat, the result might be int). Therefore, in Isabelle VDM nats become VDMNat, which are just a synonym for $\mathbb{Z}$.

Isabelle recursive functions requires a proof obligation that parameters represent a constructive and compatible pattern, and that recursive calls terminate. Nevertheless, $\mathbb{Z}$ in Isabelle is defined in terms of a pair of $\mathbb{N}$, hence recursion over $\mathbb{Z}$ are involved. Given VDM needs to represent its nat variables as VDMNat this will make VDM recursive functions in Isabelle involved as well.

To give a concrete example, we define a recursive implementation of factorial in VDM translated to Isabelle as:

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

For translation, we firt encode the implicit precondition of factorial that insists that the given parameter n is a VDMNat

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

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

```
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)›
```

Pattern compatibility and completeness is discharged with the usual proof strategy in this case. In the general (more complex recursive call cases, e.g., Ackerman's function for instance), the user might have goals to discharge.

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

Following the same strategy as before for sets, we define a relation representing the recursive and original call, respectively. The relation contains only input values that satisfy the precondition of factorial as well as the specific case which recursion occurs (e.g. (0::'a) < n).

```
abbreviation
  vdm-factorial-wf :: ‹(VDMNat × VDMNat) set›
  where
  vdm-factorial-wf ≡ { (n − 1, n) | n . 0 < n ∧ pre-vdm-factorial n }
```

To make well foundedness proof easy, we reuse an already proved well founded relation for the integers, with the relation int-ge-less-than[display] which we start from 0::′a. This is quite similar to the strategy used for finite subsets with finite-psubset[display].

```
definition
  vdm-factorial-term :: ‹(VDMNat × VDMNat) set›
  where
  ‹vdm-factorial-term ≡ (int-ge-less-than 0) ∩ vdm-factorial-wf›
```

Because int-ge-less-than is already well founded (e.g. wf (int-ge-less-than ?d)[display]), the proof for our definition is trivial.

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

To mak 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.

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

Finally, we prove termination using the previously proved lemmas using the relation. 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.

```
termination
  apply (relation ‹vdm-factorial-term›)
   apply (simp add: l-vdm-factorial-term-wf)
 by (simp add: inv-VDMNat-def pre-vdm-factorial-def vdm-factorial-term-def int-ge-less-than-def)

end
```