resources

Leo Freitas

November 1, 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 $\mathbb{N}$)).

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 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

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.

definition
  sumset-term ::(VDMNat VDMSet × VDMNat VDMSet) set where
   sumset-term ≡ finite-psubset ∩ { (s − {(SOME e . e ∈ s)}, s)| s . s ≠ {} ∧ pre-sumset s }

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]

```
  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. ⋀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?

lemma l-sumset-term-not-empty: sumset-term $\neq$ {}
  apply safe
  find-theorems elim
  apply (erule equalityE)
  find-theorems {} $\subseteq$ -
  find-theorems - $\subseteq$ {} elim
  find-theorems (- $\subseteq$ -) = -
  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)

end