CSC3323 Isabelle Tutorials

By Leo Freitas

November 2, 2015

Contents

1 Introduction

This theory file is a manual translation of the corresponding Overture VDM model. You are expected to read this document whilst playing with the theory file in Isabelle and Overture.

2 Imports

```
module NimFull
imports from IO    functions  println renamed println;
                              --printf  renamed printf;
```

```
                                      print    renamed print
  exports all
  definitions
```

We use `VDMSeq.thy`, which contains various auxiliary functions translating VDM sequences into Isabelle lists. The `While_Combinator.thy` theory provides a while-like operator for the main game play. Moreover, we are not translating the auxiliary IO functions, which are just for Overture model debugging.

3 VDM values

Values are trivial: we add them as *abbreviation*s. Notice that we would need to add invariants here about $\mathbb{N}$.

```
values

MAX_PILE: nat1 = 20;
MAX_MOV: nat1 = 3;
```

abbreviation
MAX-PILE :: *VDMNat1* **where** *MAX-PILE* $\equiv$ *20*
abbreviation
MAX-MOV :: *VDMNat1* **where** *MAX-MOV* $\equiv$ *3*

definition
inv-MAX-PILE :: $\mathbb{B}$
where
inv-MAX-PILE $\equiv$ *inv-VDMNat1 MAX-PILE*

definition
inv-MAX-MOV :: $\mathbb{B}$
where
inv-MAX-MOV $\equiv$ *inv-VDMNat1 MAX-MOV* $\wedge$ *MAX-MOV* $<$ *MAX-PILE*

Remember the implicit invariant, from requirements, that *MAX-MOV* $<$ *MAX-PILE*, otherwise a player could play to loose from the beginning. This was not in the Overture Module because we gave explicit values, which implied this invariant.

The fixing of values was just for the benefit of animating the model in overture. All that we really cared about was the axiom (given) that these constants should be $\mathbb{N}_1$, and that move limit cannot be the whole pile.

axiomatization
G-MAX-PILE :: *VDMNat*
and *G-MAX-MOV* :: *VDMNat*
where

$G\text{-}MAX\text{-}PILE > 0$
and $G\text{-}MAX\text{-}MOV > 0$
and $G\text{-}MAX\text{-}MOV < G\text{-}MAX\text{-}PILE$

Another important observation is the colour code Isabelle uses for **known**, **free** and **bound** variables. For example, in the predicate

$\forall\, e{\in}elems\ s.\ (0{::}'a) < e$

the (**black**) name *elems* is known (i.e. previously defined), *s* (**blue**) is free (i.e. externally given), and *e* (**green**) is bound (i.e. defined locally in the context of the universal quantifier).

4 VDM types

4.1 Player

```
types

-- leave fair play out of game types for simplicity;
-- include it in the game play algorithm instead
Player = <P1> | <P2> ;
```

VDM enumerated types can be declared as Isabelle data type constants. All that matters is that $P1 \neq P2$ and that those are the only values of type *Player*.

datatype $Player = P1 \mid P2$

4.2 Move

```
Move = nat1
inv m == m <= MAX_MOV;
```

We use *type-synonym* for VDM types, where type invariants must be explicitly declared as boolean-valued functions. Note in this case, we also add the invariant about $\mathbb{N}_1$, which says that $0 < m$ and is defined in theory `VDMBasic.thy` imported through `VDMSeq.thy`.

type-synonym $Move = VDMNat1$

definition
$inv\text{-}Move :: Move \Rightarrow \mathbb{B}$
where
$inv\text{-}Move\ m \equiv inv\text{-}VDMNat1\ m \wedge m \leq MAX\text{-}MOV$

4.2.1 Useful lemmas about $Move$ invariant:

Proof steps noted with "–SH" were discovered with the automated proof tool called sledgehammer. If Isabelle knows "enough" information about newly defined concepts, it often discovers proofs. Identifying what "enough" means in context is part of the challenge.

lemma *l-inv-Move-nat1* :
inv-Move m $\Longrightarrow$ *0 < m*
unfolding *inv-Move-def inv-VDMNat1-def* **by** *simp* — SH

1. every move m is $\mathbb{N}_1$:

   inv-Move ?m $\Longrightarrow$ *0 < ?m*

4.3 *sum-elems* function

Isabelle requires declaration before use, hence to define the *inv-Moves* we must have previously defined *sum-elems*.

```
functions

sum_elems: seq of Move -> nat
sum_elems(s) ==
  cases s:
    []     -> 0,
    [x]^xs -> x + sum_elems(xs)
  end
post
  -- if someone played, then sum is not zero
  s <> [] <=> RESULT > 0
measure sum_elems_measure;

sum_elems_measure: seq of Move -> nat
sum_elems_measure(s) == len s;
```

The sum of moves is defined recursively on the length of the list. Like in VDM, pattern matching is used. In Isabelle you must define a pattern for every *datatype* constructor. For lists they are empty and cons as in VDM. We also need to explicitly add the precondition about its type invariant implicitly checked by Overture. Isabelle infers a measure function automatically in most cases.

Notice that *sum-elems* operate over sequence of $Move$ rather than the type *Moves*. That is important because the invariant of $Moves$ is defined using *sum-elems*. If *sum-elems* signature involved $Moves$, its type invariant would have been called, hence leading to a loop. Overture sadly falls short of a good error message.

Isabelle does not does not check type invariants and requires declaration before use. When pre/post are not declared in Overture, we need to define them in order to ensure types are properly checked.

fun
sum-elems :: (*Move VDMSeq*) ⇒ *VDMNat*
where
sum-elems [] = *0*
| *sum-elems* (*x* # *xs*) = *x* + (*sum-elems xs*)

4.3.1 Useful lemmas

lemma *l-sum-elems-nat*:
inv-SeqElems inv-Move s ⟹ *0* ≤ *sum-elems s*
unfolding *inv-SeqElems-def*
apply (*induct s*, *simp-all*)
using *l-inv-Move-nat1* **by** *fastforce* — SH

lemma *l-sum-elems-nat1*:
inv-SeqElems inv-Move s ⟹ *s* ≠ [] ⟹ *0* < *sum-elems s*
apply (*induct s*)
apply *simp-all*
apply (*simp add*: *l-inv-SeqElems-Cons*)
apply (*erule conjE*)
apply (*frule l-inv-Move-nat1*)
apply (*frule l-sum-elems-nat*)
apply *simp*

No subgoals!

This finishes the proof but I want to have it discovered by sledgehammer.

oops

lemma *l-sum-elems-nat1*:
inv-SeqElems inv-Move s ⟹ *s* ≠ [] ⟹ *0* < *sum-elems s*
by (*smt inv-SeqElems-def l-inv-Move-nat1 l-sum-elems-nat list.pred-inject*(2) *sum-elems.elims*)

lemma *l-sum-elems-notempty*:
inv-SeqElems inv-Move s ⟹ *0* < *sum-elems s* ⟹ *s* ≠ [] **by** *auto* — SH

1. sum of elements for a sequence of $Move$ is $\mathbb{N}$:

   inv-SeqElems inv-Move ?s ⟹ *0* ≤ *sum-elems ?s*

2. sum of elements for a non empty sequence of $Move$ is $\mathbb{N}_1$:

$[\![inv\text{-}SeqElems\ inv\text{-}Move\ ?s;\ ?s \neq [\,]]\!] \Longrightarrow 0 < sum\text{-}elems\ ?s$

3. non-empty sequence when sum of elements is $\mathbb{N}_1$:

   $[\![inv\text{-}SeqElems\ inv\text{-}Move\ ?s;\ 0 < sum\text{-}elems\ ?s]\!] \Longrightarrow ?s \neq [\,]$

4.3.2 Specification

definition
pre-sum-elems :: *Move VDMSeq* $\Rightarrow \mathbb{B}$
where
pre-sum-elems s $\equiv$ *inv-SeqElems inv-Move s*

definition
post-sum-elems :: *Move VDMSeq* $\Rightarrow$ *VDMNat* $\Rightarrow \mathbb{B}$
where
post-sum-elems s RESULT $\equiv$
  inv-SeqElems inv-Move s $\wedge$ *inv-VDMNat RESULT* $\wedge$
  $(s \neq [\,] \longleftrightarrow 0 < RESULT)$

Useful properties about *sum-elems* specification.

lemma *l-pre-sum-elems*:
inv-SeqElems inv-Move s $\Longrightarrow 0 <$ *sum-elems s* $\longleftrightarrow s \neq [\,]$
using *l-sum-elems-nat1* **by** *auto* — SH

lemma *l-pre-sum-elems-sat*:
pre-sum-elems s $\Longrightarrow 0 <$ *sum-elems s* $\longleftrightarrow s \neq [\,]$
unfolding *l-sum-elems-nat1 pre-sum-elems-def* **by** (*simp add*: *l-pre-sum-elems*) — SH

These (trivial) intermediate results help us ensure that *sum-elems* specification is satisfiable by helping Isabelle sledgehammer find proofs

4.3.3 Example PO: auxiliary function satisfiability

Next, we illustrate the general PO setup for all auxiliary functions. After the translation is complete, one needs to translate proof obligations to ensure pre/post are satisfiable. The theorem layout depends on whether there is an explicit definition for the auxiliary function, given explicit definitions will determine the existential witness(es). For instance, for an implicitly defined VDM function

```
f(i: T1) r: T2
pre pre_f(i)
post post_f(i, r)
```

we need to prove this satisfiability theorem in Isabelle:

$\forall\, i.\ inv\text{-}T1\ i \longrightarrow pre\text{-}f\ i \longrightarrow (\exists\, r.\ inv\text{-}T2\ r \wedge post\text{-}f\ i\ r)$

whereas, for an explicitly defined VDM function

```
f: T1 -> T2
f(i) == expr
pre pre_f(i)
post post_f(i, RESULT)
```

we need to prove this satisfiability theorem in Isabelle:

$\forall\, i.\ inv\text{-}T1\ i \longrightarrow pre\text{-}f\ i \longrightarrow inv\text{-}T2\ expr \wedge post\text{-}f\ i\ expr$

Notice that if explicit definitions are given, there is no choice for witness for the proof obligation! That is, the commitment in the model presented by the explicit definition (*e.g. expr*) must feature in the proof. This will be particularly interesting in the proof below about *best-move*, where the general case is provable, whereas the one with the initial explicit definition of *best-move* is not. **That is, the specification is feasible for some implementation but not the one given by the explicit definition!** For instance, the theorem for the explicitly defined *sum-elems* function is:

$\forall\, s.\ inv\text{-}SeqElems\ inv\text{-}Move\ s \longrightarrow$
$\quad pre\text{-}sum\text{-}elems\ s \longrightarrow post\text{-}sum\text{-}elems\ s\ (sum\text{-}elems\ s)$

That is, given any valid input value (*inv-SeqElems inv-Move ms*), if the pre condition holds, so ought to hold the post condition. We use a definition to declare such statements as conjectures and then try to prove them as theorems.

definition
 PO-sum-elems-sat-obl :: $\mathbb{B}$
where
 $PO\text{-}sum\text{-}elems\text{-}sat\text{-}obl \equiv \forall\ s\ .\ inv\text{-}SeqElems\ inv\text{-}Move\ s \longrightarrow$
 $\quad pre\text{-}sum\text{-}elems\ s \longrightarrow (\exists\ r\ .\ post\text{-}sum\text{-}elems\ s\ r)$

definition
 PO-sum-elems-sat-exp-obl :: $\mathbb{B}$
where
 $PO\text{-}sum\text{-}elems\text{-}sat\text{-}exp\text{-}obl \equiv \forall\ s\ .\ inv\text{-}SeqElems\ inv\text{-}Move\ s \longrightarrow$
 $\quad pre\text{-}sum\text{-}elems\ s \longrightarrow post\text{-}sum\text{-}elems\ s\ (sum\text{-}elems\ s)$

We first prove the goal manually, followed by sledgehammer discovered proofs, given the lemmas created below.

theorem *PO-sum-elems-sat-obl*
unfolding *PO-sum-elems-sat-obl-def post-sum-elems-def pre-sum-elems-def*
apply (*intro allI impI conjI*)

apply (*rule-tac x=sum-elems s* **in** *exI*, *intro conjI*, *assumption*)
apply (*simp add*: *l-sum-elems-nat inv-VDMNat-def*)
by (*simp add*: *l-pre-sum-elems*)

theorem *PO-sum-elems-sat-obl*
by (*metis PO-sum-elems-sat-obl-def inv-VDMNat-def*
l-pre-sum-elems-sat leD linear post-sum-elems-def) — SH

theorem *PO-sum-elems-sat-exp-obl*
by (*simp add*: *PO-sum-elems-sat-exp-obl-def inv-VDMNat-def*
l-pre-sum-elems l-sum-elems-nat post-sum-elems-def) — SH

4.4 Moves

```
Moves = seq of Move
inv s ==
  -- you can never move beyond what's in the pile
  sum_elems(s) <= MAX_PILE
  and
  -- last move is always 1, when moves are present, at the end of
      the game
  (sum_elems(s) = MAX_PILE => s(len s) = 1)
```

Because *Moves* depends on *sum-elems*, it must be declared after it. Moreover, its invariant uses sequence application ($s(lens)$), which will need adjustment (see values example below). value and lemma commands can be used to explore the space of options and whether the expression you type does what you want.

In Isabelle, list application is defined as *s* ! *i*. But remember that Isabelle's lists are indexed from 0, whereas VDM sequences are indexed from 1. Check our version of sequence application operator (e.g. in VDM $s(x)$, in Isabelle *applyVDMSeq s x*), particularly when called outside the bounds of the sequence.

value [*a*,*b*] ! *0*
value [*a*,*b*] ! *1*
value [*a*,*b*] ! *2*
value [*a*,*b*] ! *nat* (*len* [*a*,*b*])
value [*a*,*b*] ! *nat* (*len* [*a*,*b*] − *1*)
value *applyVDMSeq* [*a*,*b*] (*len* [*a*,*b*])

type-synonym *Moves* = *Move VDMSeq*

definition
inv-Moves :: *Moves* ⇒ $\mathbb{B}$
where
inv-Moves s ≡
inv-SeqElems inv-Move s ∧
pre-sum-elems s ∧

$(let\ r = sum\text{-}elems\ s\ in$
$post\text{-}sum\text{-}elems\ s\ r \wedge$
$r \leq MAX\text{-}PILE \wedge$
$(r = MAX\text{-}PILE \longrightarrow applyVDMSeq\ s\ (len\ s) = 1))$

Finally, as the type invariant depends on another function, we need to ensure its dependent function(s) (e.g. *sum-elems*) precondition(s) features in it. Sometimes value does not work[1]. Then, lemma can be used.

value *inv-Move 2*
value *inv-Moves* [*2*,*20*]
value *sum-elems* [*2*,*3*,*4*]
value *inv-SeqElems inv-Move* [*2*,*3*,*2*,*1*]
value *inv-SeqElems inv-Move* [*2*,*3*,*4*,*1*]

5 VDM auxiliary functions

5.1 *who-plays-next*

```
-- isabelle requires declaration before use!
isFirst: Player -> bool
isFirst(p) == p = <P1>;

-- assumes <P1> is the first player
who_plays_next: Moves -> Player
who_plays_next(ms) ==
  if len ms mod 2 = 0 then <P1> else <P2>
pre isFirst(<P1>);
```

definition
$who\text{-}plays\text{-}next :: Moves \Rightarrow Player$
where
$who\text{-}plays\text{-}next\ ms \equiv (if\ (len\ ms)\ mod\ 2 = 0\ then\ P1\ else\ P2)$

definition
$isFirst :: Player \Rightarrow \mathbb{B}$
where
$isFirst\ p \equiv p = P1$

Given there is no pre/post for *isFirst*, and no type invariants to check, modelling pre/post is optional. **Make sure you know when this is okay!**

5.1.1 Specification

definition
$pre\text{-}who\text{-}plays\text{-}next :: Moves \Rightarrow \mathbb{B}$

[1] Like in Overture, in some circumstances Isabelle does not know how to evaluate expressions

where
pre-who-plays-next ms $\equiv$ *inv-Moves ms* $\wedge$ *isFirst P1*

definition
post-who-plays-next :: *Moves* $\Rightarrow$ *Player* $\Rightarrow$ $\mathbb{B}$
where
post-who-plays-next ms RESULT $\equiv$ *inv-Moves ms*

5.1.2 Satisfiability PO

definition
PO-who-plays-next-sat-obl :: $\mathbb{B}$
where
PO-who-plays-next-sat-obl $\equiv \forall$ *s* . *inv-Moves s* $\longrightarrow$
pre-who-plays-next s $\longrightarrow$ ($\exists$ *r* . *post-who-plays-next s r*)

theorem *PO-who-plays-next-sat-obl*
by (*simp add*: *PO-who-plays-next-sat-obl-def post-who-plays-next-def*) — SH

definition
PO-who-plays-next-sat-exp-obl :: $\mathbb{B}$
where
PO-who-plays-next-sat-exp-obl $\equiv \forall$ *s* . *inv-Moves s* $\longrightarrow$
pre-who-plays-next s $\longrightarrow$ *post-who-plays-next s* (*who-plays-next s*)

theorem *PO-who-plays-next-sat-exp-obl*
by (*simp add*: *PO-who-plays-next-sat-exp-obl-def post-who-plays-next-def*) — SH

5.2 *fair-play*

```
fair_play: Player * Moves -> bool
fair_play(p, ms) == p = who_plays_next(ms);
```

Notice that in Isabelle, we get curried definitions (e.g. *fair-play* is called as *fair-play p ms*) for VDM functions with multiple parameters.

definition
fair-play :: *Player* $\Rightarrow$ *Moves* $\Rightarrow$ $\mathbb{B}$
where
fair-play p ms $\equiv$ *p* = *who-plays-next ms*

5.2.1 Specification

definition
pre-fair-play :: *Player* $\Rightarrow$ *Moves* $\Rightarrow$ $\mathbb{B}$
where
pre-fair-play p ms $\equiv$ *inv-Moves ms* $\wedge$ *pre-who-plays-next ms*

definition
post-fair-play :: *Player* ⇒ *Moves* ⇒ $\mathbb{B}$ ⇒ $\mathbb{B}$
where
post-fair-play p ms RESULT ≡ *inv-Moves ms* ∧
pre-who-plays-next ms ∧ *post-who-plays-next ms p*

5.2.2 Satisfiability PO

definition
PO-fair-play-sat-obl :: $\mathbb{B}$
where
PO-fair-play-sat-obl ≡ ∀ *s p* . *inv-Moves s* ⟶
pre-fair-play p s ⟶ (∃ *r* . *post-fair-play p s r*)

theorem *PO-fair-play-sat-obl*
using *PO-fair-play-sat-obl-def post-fair-play-def*
post-who-plays-next-def pre-fair-play-def **by** *auto* — SH

definition
PO-fair-play-sat-exp-obl :: $\mathbb{B}$
where
PO-fair-play-sat-exp-obl ≡ ∀ *s p* . *inv-Moves s* ⟶
pre-fair-play p s ⟶ *post-fair-play p s* (*fair-play p s*)

theorem *PO-fair-play-sat-exp-obl*
using *PO-fair-play-sat-exp-obl-def post-fair-play-def*
post-who-plays-next-def pre-fair-play-def **by** *auto* — SH

5.3 *moves-left*

```
moves_left: Moves -> nat
moves_left(ms) == MAX_PILE - sum_elems(ms);
```

definition
moves-left :: *Moves* ⇒ *VDMNat*
where
moves-left ms ≡ (*MAX-PILE* − *sum-elems ms*)

5.3.1 Specification

definition
pre-moves-left :: *Moves* ⇒ $\mathbb{B}$
where
pre-moves-left ms ≡ *inv-Moves ms* ∧ *pre-sum-elems ms*

I label initial versions of specification later found to be problematic through failed proof with a trailing 0. I keep versions here for the sake of exposition of how mistakes can happen and what to do about them. The difference is that the first version calls *post-sum-elems* with the wrong *RESULT*.

Unfortunately, that necessarily complicates the underlying explanation. Remember that you are expected to read this document whilst playing with the theory file in Isabelle and Overture.

definition
post-moves-left0 :: *Moves* $\Rightarrow$ *VDMNat* $\Rightarrow$ $\mathbb{B}$
where
post-moves-left0 ms RESULT $\equiv$
inv-Moves ms $\wedge$ *inv-VDMNat RESULT* $\wedge$
pre-sum-elems ms $\wedge$
post-sum-elems ms RESULT

definition
post-moves-left :: *Moves* $\Rightarrow$ *VDMNat* $\Rightarrow$ $\mathbb{B}$
where
post-moves-left ms RESULT $\equiv$
inv-Moves ms $\wedge$ *inv-VDMNat RESULT* $\wedge$
pre-sum-elems ms $\wedge$
post-sum-elems ms (*sum-elems ms*)

5.3.2 Satisfiability PO

definition
PO-moves-left-sat-obl0 :: $\mathbb{B}$
where
PO-moves-left-sat-obl0 $\equiv \forall$ *s* . *inv-Moves s* $\longrightarrow$
pre-moves-left s $\longrightarrow$ ($\exists$ *r* . *post-moves-left0 s r*)

theorem *PO-moves-left-sat-obl0*
by (*meson PO-moves-left-sat-obl0-def inv-Moves-def*
post-moves-left0-def post-sum-elems-def) — SH

definition
PO-moves-left-sat-obl :: $\mathbb{B}$
where
PO-moves-left-sat-obl $\equiv \forall$ *s* . *inv-Moves s* $\longrightarrow$
pre-moves-left s $\longrightarrow$ ($\exists$ *r* . *post-moves-left s r*)

theorem *PO-moves-left-sat-obl*
by (*meson PO-moves-left-sat-obl-def inv-Moves-def*
post-moves-left-def post-sum-elems-def) — SH

definition
PO-moves-left-sat-exp-obl :: $\mathbb{B}$
where
PO-moves-left-sat-exp-obl $\equiv \forall$ *s* . *inv-Moves s* $\longrightarrow$

pre-moves-left s $\longrightarrow$ *post-moves-left s* (*moves-left s*)

theorem *PO-moves-left-sat-exp-obl*
using *PO-moves-left-sat-exp-obl-def inv-Moves-def inv-VDMNat-def moves-left-def post-moves-left-def* **by** *fastforce* — SH

5.4 *play-move*

```
play_move: Player * Move * Moves -> Moves
play_move(p, m, s) == s ^ [m]
pre
  -- cannot play to loose, but at the end
  moves_left(s) <> 1 => m < moves_left(s)

  and
  --there must be something to be played
  moves_left(s) > 0

  and
  -- encodes fairness: if even no moves, then it must be <P1>'s
      turn
  fair_play(p, s)
post
  -- you play something = implicitly true by the inv of Move
  sum_elems(s) < sum_elems(RESULT)
  and
  sum_elems(s) + m = sum_elems(RESULT)
```

definition
play-move :: *Player* $\Rightarrow$ *Move* $\Rightarrow$ *Moves* $\Rightarrow$ *Moves*
where
play-move p m s $\equiv$ *s* @ [*m*]

5.4.1 Specification

definition
pre-play-move0 :: *Player* $\Rightarrow$ *Move* $\Rightarrow$ *Moves* $\Rightarrow$ $\mathbb{B}$
where
pre-play-move0 p m s $\equiv$
inv-Move m $\wedge$ *inv-Moves s* $\wedge$ *pre-moves-left s* $\wedge$ *pre-fair-play p s* $\wedge$
post-fair-play p s (*fair-play p s*) $\wedge$
(*moves-left s* $\neq$ *1* $\longrightarrow$ *m* $<$ *moves-left s*) $\wedge$
0 $<$ *moves-left s* $\wedge$ *fair-play p s*

definition
pre-play-move :: *Player* $\Rightarrow$ *Move* $\Rightarrow$ *Moves* $\Rightarrow$ $\mathbb{B}$
where
pre-play-move p m s $\equiv$
inv-Move m $\wedge$ *inv-Moves s* $\wedge$ *pre-moves-left s* $\wedge$ *pre-fair-play p s* $\wedge$

$post\text{-}fair\text{-}play\ p\ s\ (fair\text{-}play\ p\ s) \wedge$
$fair\text{-}play\ p\ s \wedge$
$m \leq moves\text{-}left\ s \wedge$
$(moves\text{-}left\ s = m \longrightarrow m = 1)$

definition
$post\text{-}play\text{-}move :: Player \Rightarrow Move \Rightarrow Moves \Rightarrow Moves \Rightarrow \mathbb{B}$
where
$post\text{-}play\text{-}move\ p\ m\ s\ RESULT \equiv$
$inv\text{-}Move\ m \wedge inv\text{-}Moves\ s \wedge inv\text{-}Moves\ RESULT \wedge$
$pre\text{-}sum\text{-}elems\ s \wedge pre\text{-}sum\text{-}elems\ RESULT \wedge$
$post\text{-}sum\text{-}elems\ s\ (sum\text{-}elems\ s) \wedge post\text{-}sum\text{-}elems\ RESULT\ (sum\text{-}elems\ RESULT) \wedge$
$sum\text{-}elems\ s < sum\text{-}elems\ RESULT \wedge$
$sum\text{-}elems\ s + m = sum\text{-}elems\ RESULT$

5.4.2 Satisfiability PO

This PO is rather involved and will be discussed later in the text.

5.5 *will-first-player-win*

```
will_first_player_win: () -> bool
will_first_player_win() == (MAX_PILE - 1) mod (MAX_MOV + 1) <> 0;
```

VDM parameterless functions are just like constants of the result type. Be careful with expressions like $x\ (-\ 1)$ and $x - 1$: the former applies the function x to the parameter -1, whereas the second applies the subtraction function to two parameters x and 1. Think of negative numbers as a unary function.

definition
$will\text{-}first\text{-}player\text{-}win :: \mathbb{B}$
where
$will\text{-}first\text{-}player\text{-}win \equiv (MAX\text{-}PILE - 1)\ mod\ (MAX\text{-}MOV + 1) \neq 0$

5.5.1 Specification

definition
$pre\text{-}will\text{-}first\text{-}player\text{-}win :: \mathbb{B}$
where
$pre\text{-}will\text{-}first\text{-}player\text{-}win \equiv inv\text{-}MAX\text{-}PILE$

The precondition is needed to avoid applying the modulo operator to negative numbers

5.5.2 Satisfiability PO

definition

PO-will-first-player-win-sat-obl :: $\mathbb{B}$
where
PO-will-first-player-win-sat-obl $\equiv$
pre-will-first-player-win $\longrightarrow$ ($\exists$ *r* . *r*)

theorem *PO-will-first-player-win-sat-obl*
using *PO-will-first-player-win-sat-obl-def* **by** *auto* — SH

definition
PO-will-first-player-win-sat-exp-obl :: $\mathbb{B}$
where
PO-will-first-player-win-sat-exp-obl $\equiv$
pre-will-first-player-win $\longrightarrow$ *will-first-player-win*

theorem *PO-will-first-player-win-sat-exp-obl*
using *PO-will-first-player-win-sat-exp-obl-def*
pre-will-first-player-win-def *will-first-player-win-def* **by** *simp* — SH

5.6 *who-won-invariant*

```
-- invariant for whoever won: last player looses by taking 1
-- even seq means second player; odd seq means firs player
who_won_invariant: Player * Moves -> bool
who_won_invariant(winner, moves) ==
  -- all moves played, including last
  moves_left(moves) = 0
  and
  -- if the winner plays next, then the last guy lost, given there
       are no more moves left
  winner = who_plays_next(moves)
  -- assuming perfect play?
  and
  will_first_player_win() => isFirst(winner)
```

definition
who-won-invariant :: *Player* $\Rightarrow$ *Moves* $\Rightarrow$ $\mathbb{B}$
where
who-won-invariant winner moves $\equiv$
moves-left moves = *0*
$\wedge$
winner = *who-plays-next moves*
$\wedge$
will-first-player-win $\longrightarrow$ *isFirst winner*

5.6.1 Specification

definition
pre-who-won-invariant :: *Player* $\Rightarrow$ *Moves* $\Rightarrow$ $\mathbb{B}$

where
pre-who-won-invariant winner moves $\equiv$
inv-Moves moves $\wedge$ *pre-moves-left moves* $\wedge$
pre-will-first-player-win $\wedge$ *pre-who-plays-next moves*

definition
post-who-won-invariant :: *Player* $\Rightarrow$ *Moves* $\Rightarrow$ $\mathbb{B} \Rightarrow \mathbb{B}$
where
post-who-won-invariant winner moves RESULT $\equiv$
inv-Moves moves $\wedge$ *pre-moves-left moves* $\wedge$
post-moves-left moves (*moves-left moves*) $\wedge$
pre-will-first-player-win $\wedge$
pre-who-plays-next moves $\wedge$ *post-who-plays-next moves winner*

5.6.2 Satisfiability PO

definition
PO-who-won-invariant-sat-obl :: $\mathbb{B}$
where
PO-who-won-invariant-sat-obl $\equiv \forall$ *s p* . *inv-Moves s* $\longrightarrow$
pre-who-won-invariant p s $\longrightarrow$ ($\exists$ *r* . *post-who-won-invariant p s r*)

theorem *PO-who-won-invariant-sat-obl*
unfolding *PO-who-won-invariant-sat-obl-def post-who-won-invariant-def*
pre-who-won-invariant-def
using *inv-Moves-def inv-VDMNat-def moves-left-def*
post-moves-left-def post-who-plays-next-def **by** *fastforce* — SH

definition
PO-who-won-invariant-sat-exp-obl :: $\mathbb{B}$
where
PO-who-won-invariant-sat-exp-obl $\equiv \forall$ *s p* . *inv-Moves s* $\longrightarrow$
pre-who-won-invariant p s $\longrightarrow$ *post-who-won-invariant p s* (*who-won-invariant p s*)

theorem *PO-who-won-invariant-sat-exp-obl*
unfolding *PO-who-won-invariant-sat-exp-obl-def post-who-won-invariant-def*
pre-who-won-invariant-def
using *inv-Moves-def inv-VDMNat-def moves-left-def*
post-moves-left-def post-who-plays-next-def **by** *fastforce* — SH

5.7 *first-player*

```
first_player: () -> Player
first_player() == if isFirst(<P1>) then <P1> else <P2>
post isFirst(RESULT);
```

definition

first-player :: *Player*
where
first-player ≡ (*if isFirst P1 then P1 else P2*)

5.7.1 Specification

definition
post-first-player :: *Player* ⇒ $\mathbb{B}$
where
post-first-player RESULT ≡ *isFirst RESULT*

5.7.2 Satisfiability PO

definition
PO-first-player-sat-obl :: $\mathbb{B}$
where
PO-first-player-sat-obl ≡ (∃ *r* . *post-first-player r*)

theorem *PO-first-player-sat-obl*
unfolding *PO-first-player-sat-obl-def post-first-player-def*
by (*simp add*: *isFirst-def*) — SH

definition
PO-first-player-sat-exp-obl :: $\mathbb{B}$
where
PO-first-player-sat-exp-obl ≡ *post-first-player first-player*

theorem *PO-first-player-sat-exp-obl*
unfolding *PO-first-player-sat-exp-obl-def post-first-player-def*
first-player-def
by (*simp add*: *isFirst-def*) — SH

5.8 *first-player-inds*

```
first_player_inds: Moves -> set of nat1
first_player_inds(ms) == { i | i in set inds ms & i mod 2 <> 0 }
post RESULT subset inds ms;
```

definition
first-player-inds :: *Moves* ⇒ $\mathbb{N}$ *set*
where
first-player-inds ms ≡ { *i* | *i* . *i* ∈ *inds-as-nat ms* ∧ *i mod 2* ≠ *0* }

Again, value and lemma commands can be used to explore the space of options desired. Whenever value fails (see commented expression in theory file), that is because Isabelle does not know how to enumerate the expression, like in certain circumstances Overture cannot execute models. For that, we can use lemmas and simple proofs. The "proof" here is really debugging as we do not know whether the expected expression means what we want/intend, hence the *oops* command.

value $\{i\ .\ i \in \{(0::int),1,2,3\}\}$
value $\{(i,i) \mid i\ .\ i \in \{(0::int),1,2,3\}\ \}$

lemma $A = \{i\ .\ i \in \{(0::int),1,2,3\} \mid i < 2\}$ **apply** *simp* **oops**
lemma $A = \{i \mid i\ .\ i \in \{(0::int),1,2,3\} \wedge i < 2\}$ **apply** *simp* **oops**
lemma $\{0,1\} = \{i \mid i\ .\ i \in \{(0::int),1,2,3\} \wedge i < 2\}$ **apply** *auto* **done**

5.8.1 Specification

definition
pre-first-player-inds :: *Moves* ⇒ 𝔹
where
pre-first-player-inds ms ≡ *inv-Moves ms*

definition
post-first-player-inds :: *Moves* ⇒ ℕ *VDMSet* ⇒ 𝔹
where
post-first-player-inds ms RESULT ≡ *inv-Moves ms* ∧
inv-SetElems nat1 RESULT ∧ *RESULT* ⊆ *inds-as-nat ms*

5.8.2 Satisfiability PO

definition
PO-first-player-inds-sat-obl :: 𝔹
where
PO-first-player-inds-sat-obl ≡ ∀ *s* . *inv-Moves s* ⟶
pre-first-player-inds s ⟶ (∃ *r* . *post-first-player-inds s r*)

theorem *PO-first-player-inds-sat-obl*
using *PO-first-player-inds-sat-obl-def inv-SetElems-def post-first-player-inds-def* **by** *auto*
— SH

definition
PO-first-player-inds-sat-exp-obl :: 𝔹
where
PO-first-player-inds-sat-exp-obl ≡ ∀ *s* . *inv-Moves s* ⟶
pre-first-player-inds s ⟶ *post-first-player-inds s* (*first-player-inds s*)

lemma *l-first-player-inds-nat1*:
inv-Moves s ⟹ *inv-SetElems nat1* (*first-player-inds s*)
unfolding *first-player-inds-def inds-as-nat-def len-def inv-SetElems-def nat1G0*
by *simp* — SH

lemma *l-first-player-inds-within-inds*:
first-player-inds s ⊆ *inds-as-nat s*
unfolding *first-player-inds-def inds-as-nat-def len-def inv-SetElems-def nat1G0*
find-theorems - ⊆ - *intro*

apply (*rule subsetI*)
by *simp*

theorem *PO-first-player-inds-sat-exp-obl*
unfolding *PO-first-player-inds-sat-exp-obl-def post-first-player-inds-def pre-first-player-inds-def*
apply *simp*
apply (*intro allI impI conjI*)
apply (*simp add*: *l-first-player-inds-nat1*)
by (*simp add*: *l-first-player-inds-within-inds*)

5.9 *moves-of*

```
moves_of: Moves * bool -> seq of Move
moves_of(ms, first) ==
  let idxs = first_player_inds(ms) in
      [ ms(i) | i in set  if (first) then idxs else inds ms \ idxs
           ]
```

Isabelle does not allow for sets to bound variables used in list comprehension generators. That means either you need to use a sequence as a generator, or transform a set into a sorted list (by the ordering of the underlying elements). If the set of elements does not have a defined order sorting will fail. Also, *sorted-list-of-set* can lead to complicated proofs. Avoid if possible. I show it here in case you are keen on using it.

value $[\, [a,b,c] \mathbin{!} i \,.\, i \leftarrow [2,1,0]]$

value $[\, [a,b,c] \mathbin{!} i \,.\, i \leftarrow \textit{sorted-list-of-set}\ (\{a,b,c\})]$

definition
moves-of :: *Moves* $\Rightarrow \mathbb{B} \Rightarrow$ *Move VDMSeq*
where
moves-of ms first $\equiv$
(*let idxs* = *first-player-inds ms in*
[*ms* ! *i* . *i* $\leftarrow$ *sorted-list-of-set* (*if first then idxs else* (*inds-as-nat ms* $-$ *idxs*))])

5.9.1 Specification

definition
pre-moves-of :: *Moves* $\Rightarrow \mathbb{B} \Rightarrow \mathbb{B}$
where
pre-moves-of ms first $\equiv$ *inv-Moves ms* $\wedge$ *pre-first-player-inds ms*

definition
post-moves-of :: *Moves* $\Rightarrow \mathbb{B} \Rightarrow$ *Move VDMSeq* $\Rightarrow \mathbb{B}$
where
post-moves-of ms first RESULT $\equiv$

inv-Moves ms $\wedge$ *inv-SeqElems inv-Move RESULT* $\wedge$
pre-first-player-inds ms $\wedge$ *post-first-player-inds ms* (*first-player-inds ms*)

5.9.2 Satisfiability PO

definition
PO-moves-of-sat-obl :: $\mathbb{B}$
where
PO-moves-of-sat-obl $\equiv \forall$ *s f* . *inv-Moves s* $\longrightarrow$
pre-moves-of s f $\longrightarrow$ ($\exists$ *r* . *post-moves-of s f r*)

theorem *PO-moves-of-sat-obl*
unfolding *PO-moves-of-sat-obl-def post-moves-of-def pre-moves-of-def*
apply (*intro allI impI conjI*, *elim conjE*)
apply (*rule-tac x=moves-of s True* **in** *exI*)
apply *simp* **oops**

definition
PO-moves-of-sat-exp-obl :: $\mathbb{B}$
where
PO-moves-of-sat-exp-obl $\equiv \forall$ *s f* . *inv-Moves s* $\longrightarrow$
pre-moves-of s f $\longrightarrow$ *post-moves-of s f* (*moves-of s f*)

lemma *l-moves-of-move*:
inv-Moves ms $\Longrightarrow$ *inv-SeqElems inv-Move* (*moves-of ms f*)
unfolding *moves-of-def Let-def*
apply *simp*
apply (*intro conjI impI*)
unfolding *inv-SeqElems-def*
oops

theorem *PO-moves-of-sat-exp-obl*
unfolding *PO-moves-of-sat-exp-obl-def post-moves-of-def pre-moves-of-def*
apply *simp*
unfolding *post-first-player-inds-def pre-first-player-inds-def*
apply *simp*
apply (*intro allI impI conjI*)

defer
apply (*simp add*: *l-first-player-inds-nat1*) — SH
apply (*simp add*: *l-first-player-inds-within-inds*) — SH
oops

5.10 *best-move*

```
best_move: Moves -> nat
best_move(moves) == (moves_left(moves) - 1) mod (MAX_MOV + 1);
post RESULT <= moves_left(moves);
```

definition
$\textit{best-move} :: \textit{Moves} \Rightarrow \textit{VDMNat}$
where
$\textit{best-move moves} \equiv ((\textit{moves-left moves}) - \textit{1})\ \textit{mod}\ (\textit{MAX-MOV} + \textit{1})$

5.10.1 Specification

Here I explore a few versions of the specification, first the original one, which was shown to be mistaken after proofs below. The first precondition misses the fact $\textit{0} < \textit{moves-left ms}$, which prevents modulo arithmetic over negative numbers, whereas the first post condition used the wrong specification of *post-moves-left0*.

definition
$\textit{pre-best-move0} :: \textit{Moves} \Rightarrow \mathbb{B}$
where
$\textit{pre-best-move0 ms} \equiv \textit{inv-Moves ms} \wedge \textit{pre-moves-left ms}$

definition
$\textit{post-best-move0} :: \textit{Moves} \Rightarrow \textit{VDMNat} \Rightarrow \mathbb{B}$
where
$\textit{post-best-move0 ms RESULT} \equiv$
$\quad \textit{inv-Moves ms} \wedge \textit{inv-VDMNat RESULT} \wedge$
$\quad \textit{pre-moves-left ms} \wedge \textit{post-moves-left0 ms}\ (\textit{moves-left ms}) \wedge$
$\quad \textit{RESULT} \leq \textit{moves-left ms}$

definition
$\textit{pre-best-move} :: \textit{Moves} \Rightarrow \mathbb{B}$
where
$\textit{pre-best-move ms} \equiv \textit{inv-Moves ms} \wedge \textit{pre-moves-left ms} \wedge \textit{0} < \textit{moves-left ms}$

definition
$\textit{post-best-move} :: \textit{Moves} \Rightarrow \textit{VDMNat} \Rightarrow \mathbb{B}$
where
$\textit{post-best-move ms RESULT} \equiv$
$\quad \textit{inv-Moves ms} \wedge \textit{inv-VDMNat RESULT} \wedge$
$\quad \textit{pre-moves-left ms} \wedge \textit{post-moves-left ms}\ (\textit{moves-left ms}) \wedge$
$\quad \textit{RESULT} \leq \textit{moves-left ms}$

5.10.2 Satisfiability PO

After the translation is complete, one needs to create proof obligations to ensure pre/post are satisfiable. For instance, the theorem layout for *best-move* is:

$$\forall\, \textit{ms}.\ \textit{inv-Moves ms} \longrightarrow \textit{pre-best-move ms} \longrightarrow (\exists\, r.\ \textit{post-best-move ms}\ r)$$

We use a definition to declare the theorem and then prove it. Again, I show the versions I went through, and the process of discovery of the correct one. **This is very important**, and is very likely to happen to your model/translation to Isabelle.

The objective is that the proof is *True* meaning the operation is satisfiable with respect to its specification. Next we show the various proof attempts for the PO conjecture.

1 Naive attempt: layered expansion followed by simplification.

definition
PO-best-move-sat-obl0 :: $\mathbb{B}$
where
PO-best-move-sat-obl0 $\equiv \forall$ *ms* . *inv-Moves ms* $\longrightarrow$
pre-best-move0 ms $\longrightarrow$ ($\exists$ *r* . *post-best-move0 ms r*)

theorem *PO-best-move-sat-obl0*
unfolding *PO-best-move-sat-obl0-def*
pre-best-move0-def post-best-move0-def
apply *simp*
unfolding *pre-moves-left-def post-moves-left0-def*
apply *simp*
unfolding *pre-sum-elems-def post-sum-elems-def*
apply *simp*
unfolding *inv-VDMNat-def*
apply *auto*
apply (*rule-tac x=0* **in** *exI*, *intro conjI*, *simp-all*)

1. $\bigwedge$*ms*. ⟦*inv-Moves ms*; *inv-SeqElems inv-Move ms*⟧ $\Longrightarrow$ *0* $\leq$ *moves-left ms*
2. $\bigwedge$*ms*. ⟦*inv-Moves ms*; *inv-SeqElems inv-Move ms*⟧
$\Longrightarrow$ (*ms* $\neq$ []) = (*0* < *moves-left ms*)
3. $\bigwedge$*ms*. ⟦*inv-Moves ms*; *inv-SeqElems inv-Move ms*⟧ $\Longrightarrow$ *0* $\leq$ *moves-left ms*

Missing cases where we cannot make progress suggest we need lemmas on *0* $\leq$ *moves-left ms*. There is also an error: *moves-left ms* = *0* and yet *ms* $\neq$ []! We will need to change to *post-moves-left* from *post-moves-left0*.

oops

The simplistic strategy of expanding and simplifying does not work here. We need intermediate results to help Isabelle finish the proof. That means, being creative about adequate auxiliary lemmas.

lemma *l-moves-left-nat*:
inv-Moves ms $\Longrightarrow$ *0* $\leq$ *moves-left ms*
unfolding *moves-left-def inv-Moves-def Let-def* **by** *simp*

lemma *l-moves-left-nat1*:
inv-Moves ms $\Longrightarrow$ *0* < *moves-left ms*
apply (*induct ms*)
unfolding *moves-left-def*
apply *simp-all*

1. $\bigwedge a\ ms.$
$[\![inv\text{-}Moves\ ms \Longrightarrow 0 < MAX\text{-}PILE - sum\text{-}elems\ ms;\ inv\text{-}Moves\ (a \# ms)]\!]$
$\Longrightarrow 0 < MAX\text{-}PILE - (a + sum\text{-}elems\ ms)$

Missing lemma about *inv-Moves* $(x \# xs)$ distributing over list append.

oops

lemma *l-inv-Moves-Cons*:
inv-Moves $(x \# xs) = (inv\text{-}Move\ x \wedge inv\text{-}Moves\ xs)$
apply (*intro iffI conjI*)

1. *inv-Moves* $(x \# xs) \Longrightarrow inv\text{-}Move\ x$
2. *inv-Moves* $(x \# xs) \Longrightarrow inv\text{-}Moves\ xs$
3. $inv\text{-}Move\ x \wedge inv\text{-}Moves\ xs \Longrightarrow inv\text{-}Moves\ (x \# xs)$

Let us split the work again into lemmas for each subgoal to help sledgehammer!

oops

lemma *l-inv-Moves-Hd*:
inv-Moves $(x \# xs) \Longrightarrow inv\text{-}Move\ x$
unfolding *inv-Moves-def* **by** (*simp add*: *l-inv-SeqElems-Cons*) — SH

lemma *l-inv-Moves-Tl*:
inv-Moves $(x \# xs) \Longrightarrow inv\text{-}Moves\ xs$
unfolding *inv-Moves-def*
apply (*intro conjI*)
apply (*simp add*: *l-inv-SeqElems-Cons*) — SH
apply (*simp add*: *l-inv-SeqElems-Cons pre-sum-elems-def*) — SH
unfolding *Let-def*
apply (*elim conjE*, *intro conjI*)
apply (*simp add*: *post-sum-elems-def inv-SeqElems-def*
inv-VDMNat-def l-pre-sum-elems l-sum-elems-nat
pre-sum-elems-def) — SH
apply (*simp add*: *l-inv-SeqElems-Cons*, *elim conjE*)
apply (*cut-tac l-inv-Move-nat1*, *simp-all*)
apply (*intro impI*, *simp*)
apply (*simp add*: *l-inv-SeqElems-Cons l-inv-Moves-Hd*)
apply (*erule conjE*)
by (*frule l-inv-Move-nat1*, *simp*) — SH

lemma *l-inv-Moves-Cons*:
inv-Moves $(x \# xs) = (inv\text{-}Move\ x \wedge inv\text{-}Moves\ xs)$
apply (*rule iffI*)
using *l-inv-Moves-Hd l-inv-Moves-Tl* **apply** *blast* — SH
apply (*elim conjE*)
unfolding *inv-Moves-def post-sum-elems-def Let-def*
apply (*elim conjE*, *intro conjI*, *simp-all*)
apply (*simp add*: *l-inv-SeqElems-Cons*) — SH

apply (*simp add*: *l-inv-SeqElems-Cons pre-sum-elems-def*) — SH
apply (*simp add*: *l-inv-SeqElems-Cons*) — SH
using *inv-VDMNat-def l-inv-Move-nat1* **apply** *fastforce* — SH
using *l-inv-Move-nat1* **apply** *fastforce* — SH

Goals not provable when $sum\text{-}elems\ xs = MAX\text{-}PILE$, because $inv\text{-}Move\ x$ enforce $0 < x$

oops

Lemmas proved as a result of first attempt:

1.a *moves-left s* is $\mathbb{N}$ for valid moves

$inv\text{-}Moves\ ?ms \Longrightarrow 0 \leq moves\text{-}left\ ?ms$

1.b *inv-Moves s* distributes to head of *s* for valid moves

$inv\text{-}Moves\ (?x\ \#\ ?xs) \Longrightarrow inv\text{-}Move\ ?x$

1.c *inv-Moves s* distributes to tail of *s* for valid moves

$inv\text{-}Moves\ (?x\ \#\ ?xs) \Longrightarrow inv\text{-}Moves\ ?xs$

Proof failures are useful to understand what is wrong:

1.d *moves-left s* is **not** $\mathbb{N}_1$, why?

$inv\text{-}Moves\ ms \Longrightarrow 0 < moves\text{-}left\ ms$

1.e it might **not** be possible to append to a valid move sequence, why?

$inv\text{-}Moves\ (x\ \#\ xs) = (inv\text{-}Move\ x \land inv\text{-}Moves\ xs)$

Let's see if the lemma shape is working (i.e. it will be used by Isabelle).

2 Using lemmas: layered expansion followed by simplification with lemmas.

theorem *PO-best-move-sat-obl0*

...

apply (*simp add*: *l-moves-left-nat*) — SH

Yes! The lemma discharged the first suggoal, and sledgehammer found it.

oops

Next we define the PO of *best-move* with new post condition *post-best-move*, yet with the old precondition *pre-best-move0*.

3 revised definition of *post-best-move* + using lemmas: **success?!**

definition
PO-best-move-sat-obl1 :: $\mathbb{B}$
where
PO-best-move-sat-obl1 $\equiv \forall$ *ms* . *inv-Moves ms* $\longrightarrow$ *pre-best-move0 ms* $\longrightarrow$ ($\exists$ *r* . *post-best-move ms r*)

theorem *PO-best-move-sat-obl1*

...

1. $\bigwedge ms.$ $[\![$*inv-Moves ms*; *inv-SeqElems inv-Move ms*$]\!] \Longrightarrow 0 \leq$ *moves-left ms*
2. $\bigwedge ms.$ $[\![$*inv-Moves ms*; *inv-SeqElems inv-Move ms*$]\!] \Longrightarrow 0 \leq$ *sum-elems ms*
3. $\bigwedge ms.$ $[\![$*inv-Moves ms*; *inv-SeqElems inv-Move ms*$]\!]$
   $\Longrightarrow (ms \neq []) = (0 <$ *sum-elems ms*$)$
4. $\bigwedge ms.$ $[\![$*inv-Moves ms*; *inv-SeqElems inv-Move ms*$]\!] \Longrightarrow 0 \leq$ *moves-left ms*

With the updated definition, and proved lemmas, we get different subgoals, all dischargeble by sledgehammer.

apply (*simp add*: *l-moves-left-nat*) — SH
apply (*simp add*: *l-sum-elems-nat*) — SH
using *l-sum-elems-nat1* **apply** *auto*[*1*] — SH
by (*simp add*: *l-moves-left-nat*) — SH

What is going on? We proved this, shouldn't it mean that *pre-best-move0* is okay? No because we have an explicit definition as

best-move ?moves $\equiv$ (*moves-left ?moves* $- 1$) *mod* (*MAX-MOV* $+ 1$)

We need to account for that fact and be specific about the witness, which is to blame because when *moves-left ms* $= 0$, then *best-move ms* does not work as expected. **That is, if an explicit definition is given, there is no choice for witness for the proof obligation!** Thus, the commitment in the model presented by the explicit definition must feature in the proof. From Overture, the PO has a fixed witnesses according to what the explicit definition was, and we state it in Isabelle

```
best_move: function establishes postcondition obligation @ in '
    NimFull' (./NimFull.vdmsl) at line 119:1
(forall moves:Moves & post_best_move(moves, ((moves_left(moves) -
    1) mod (MAX_MOV + 1))))
Proof Obligation 15: (Unproved)
```

To avoid mixing problems from different sources, we first try to prove the original post condition with the explicit witness in the next attempt.

4 Lemmas + explicit witness + no revision of *post-best-move*

definition
PO-best-move-sat-obl2 :: $\mathbb{B}$
where
PO-best-move-sat-obl2 $\equiv \forall$ *ms . pre-best-move0 ms* $\longrightarrow$
post-best-move0 ms $(((moves\text{-}left\ ms) - 1)\ mod\ (MAX\text{-}MOV + 1))$

theorem *PO-best-move-sat-obl2*
unfolding *PO-best-move-sat-obl2-def pre-best-move0-def post-best-move0-def*
apply (*intro allI impI conjI*, *elim conjE*, *simp-all*)
unfolding *post-moves-left0-def pre-moves-left-def post-sum-elems-def*
apply (*simp-all*)

1. $\bigwedge ms.$ *inv-Moves ms* $\wedge$ *pre-sum-elems ms* $\Longrightarrow$
   inv-VDMNat $((moves\text{-}left\ ms - 1)\ mod\ 4)$
2. $\bigwedge ms.$ *inv-Moves ms* $\wedge$ *pre-sum-elems ms* $\Longrightarrow$
   inv-VDMNat $(moves\text{-}left\ ms)$ $\wedge$
   inv-SeqElems inv-Move ms $\wedge$
   inv-VDMNat $(moves\text{-}left\ ms) \wedge (ms \neq []) = (0 < moves\text{-}left\ ms)$
3. $\bigwedge ms.$ *inv-Moves ms* $\wedge$ *pre-sum-elems ms* $\Longrightarrow$
   $(moves\text{-}left\ ms - 1)\ mod\ 4 \leq moves\text{-}left\ ms$

This suggests a trivial lemma about *inv-VDMNat* to avoid multiple goals

unfolding *inv-VDMNat-def*
apply (*simp add*: *l-moves-left-nat*) — SH
apply (*intro conjI*)
apply (*simp add*: *l-moves-left-nat*) — SH
apply (*simp add*: *pre-sum-elems-def*)— SH
apply (*simp add*: *l-moves-left-nat*) — SH

1. $\bigwedge ms.$ *inv-Moves ms* $\wedge$ *pre-sum-elems ms* $\Longrightarrow (ms \neq []) = (0 < moves\text{-}left\ ms)$
2. $\bigwedge ms.$ *inv-Moves ms* $\wedge$ *pre-sum-elems ms* $\Longrightarrow$
   $(moves\text{-}left\ ms - 1)\ mod\ 4 \leq moves\text{-}left\ ms$

The first subgoal is not provable because *moves-left ms* can be 0! We can create another lemma for the final subgoal using facts about remainder using theorem search to find $0 \leq\ ?m \Longrightarrow\ ?m\ mod\ ?k \leq\ ?m$.

find-theorems - *mod* - $\leq$ -
oops

Let us create the lemmas suggested by the previous proof.

lemma *l-inv-VDMNat-moves-left*:
inv-Moves ms $\Longrightarrow$ *inv-VDMNat* $(moves\text{-}left\ ms)$
unfolding *inv-VDMNat-def* **by** (*simp add*: *l-moves-left-nat*) — SH

lemma *l-nim-mod-prop*:
$x \geq 0 \Longrightarrow (x - (1::int))\ mod\ y \leq x$
quickcheck

This is not provable with $x = 0, y = 2$. What we want is to use it for

$0 \leq$ *moves-left s* $\Longrightarrow$ *moves-left s mod MAX-MOV* $\leq$ *moves-left s*

We need to tighten our assumptions.

oops

lemma *l-nim-mod-prop*:
$x > 0 \Longrightarrow (x - (1{::}int))$ *mod* $y \leq x$
by (*smt zmod-le-nonneg-dividend*) — SH

lemma *l-moves-left-prop*:
inv-Moves ms $\Longrightarrow$ *pre-sum-elems ms* $\Longrightarrow$ (*ms* $\neq$ []) = (*0* < *moves-left ms*)
unfolding *inv-Moves-def Let-def moves-left-def*
apply (*rule iffI*)
find-theorems - $\neq$ - *name*:*Nim*
thm *l-sum-elems-nat1*[*of ms*]
apply (*cut-tac l-sum-elems-nat1*,*simp-all*)
defer
apply (*cut-tac l-sum-elems-notempty*,*simp-all*+)

oops

Proved lemmas:

4.a No need to expand *inv-VDMNat* for *moves-left ms* result;

inv-Moves ?ms $\Longrightarrow$ *inv-VDMNat* (*moves-left ?ms*)

4.b Remainder property of Nim game.

$0 < ?x \Longrightarrow (?x - 1)$ *mod* $?y \leq ?x$

Failed lemmas:

4.c Moves left might be zero, yet ms is not empty.

(*ms* $\neq$ []) = (*0* < *moves-left ms*)

Let us try again with the new lemmas.

theorem *PO-best-move-sat-obl2*
unfolding *PO-best-move-sat-obl2-def pre-best-move0-def post-best-move0-def*
apply (*intro allI impI conjI*, *elim conjE*, *simp-all*)
unfolding *post-moves-left0-def pre-moves-left-def post-sum-elems-def*
apply (*simp-all add*: *l-inv-VDMNat-moves-left*)
unfolding *inv-VDMNat-def*
apply (*simp*, *intro conjI*)
apply (*simp add*: *pre-sum-elems-def*)— SH
defer
apply (*rule l-nim-mod-prop*)

1. $\bigwedge ms.$ *inv-Moves ms* $\wedge$ *pre-sum-elems ms* $\Longrightarrow$ *0* $<$ *moves-left ms*
2. $\bigwedge ms.$ *inv-Moves ms* $\wedge$ *pre-sum-elems ms* $\Longrightarrow$ $(ms \neq []) = (0 < \textit{moves-left ms})$

Unprovable part boils down to *moves-left ms* not being $\mathbb{N}_1$.

oops

With the new lemmas for the explicit witness proved, let us now change the post condition.

5 Revised definition *post-best-move* + lemmas + explicit witness

definition
PO-best-move-sat-obl3 :: $\mathbb{B}$
where
PO-best-move-sat-obl3 $\equiv \forall$ *ms* . *pre-best-move0 ms* $\longrightarrow$
post-best-move ms (((*moves-left ms*) $-$ *1*) *mod* (*MAX-MOV* $+$ *1*))

theorem *PO-best-move-sat-obl3*
unfolding *PO-best-move-sat-obl3-def pre-best-move0-def post-best-move-def*
apply (*intro allI impI conjI*, *elim conjE*, *simp-all*)
unfolding *post-moves-left-def pre-moves-left-def post-sum-elems-def*
apply (*simp-all add*: *l-inv-VDMNat-moves-left*)
unfolding *inv-VDMNat-def*
apply *simp*
apply (*simp add*: *inv-VDMNat-def l-pre-sum-elems l-sum-elems-nat pre-sum-elems-def*)
— SH
apply (*rule l-nim-mod-prop*)

1. $\bigwedge ms.$ *inv-Moves ms* $\wedge$ *pre-sum-elems ms* $\Longrightarrow$ *0* $<$ *moves-left ms*

From the failure, let us try and prove the missing lemma.

oops

lemma *l-moves-left-nat1*:
inv-Moves ms $\wedge$ *pre-sum-elems ms* $\Longrightarrow$ *0* $<$ *moves-left ms*
unfolding *pre-sum-elems-def moves-left-def*
apply (*induct ms*, *simp-all*, *elim conjE*)
apply (*simp add*: *l-inv-Moves-Tl l-inv-SeqElems-Cons*)
apply (*elim conjE*)
apply (*frule l-inv-Move-nat1*)
apply (*frule l-sum-elems-nat*)

Goal is *False*, yet easier to see with generalised aruments

oops

lemma $0 \leq x \Longrightarrow 0 < a \Longrightarrow 0 < y - (x{::}int) \Longrightarrow 0 < y - (a + x)$
— qc: x=0,y=1,a=1

oops

Now we see what the problem is: *best-move* is missing the precondition about *moves-left* being non-zero for the explicit witness, and leads to our final attempt.

6 Revised definitions *pre-best-move* and *post-best-move* + lemmas + explicit witness

definition
PO-best-move-sat-obl :: $\mathbb{B}$
where
PO-best-move-sat-obl ≡ ∀ *ms* . *pre-best-move ms* ⟶
post-best-move ms (((*moves-left ms*) − *1*) *mod* (*MAX-MOV* + *1*))

theorem *PO-best-move-sat-obl*
unfolding *PO-best-move-sat-obl-def pre-best-move-def post-best-move-def*
apply (*intro allI impI conjI*, *elim conjE*, *simp-all*)
unfolding *inv-VDMNat-def*
apply *simp*
unfolding *post-moves-left-def pre-moves-left-def post-sum-elems-def*
apply (*intro conjI*, *elim conjE*,*simp-all*)
apply (*simp add*: *l-inv-VDMNat-moves-left*) — SH
apply (*simp add*: *pre-sum-elems-def*) — SH
apply (*meson inv-Moves-def post-sum-elems-def*) — SH

apply (*simp add*: *l-pre-sum-elems-sat*) — SH
by (*simp add*: *l-nim-mod-prop*) — SH

Finally we managed to prove that the adjusted/corrected definition of *best-move* pre and post conditions are now appropriate and make sense with the chosen specification, as well as the explicit definition. Auxiliary lemmas help sledgehammer find proofs. This illustrates how proof ensures models are fit for purpose.

5.11 *max* and *min*

```
min: int * int -> int
min(x,y) == if (x < y) then x else y;

max: int * int -> int
max(x,y) == if (x > y) then x else y;
```

Isabelle already defines these functions and we omit them here.

5.12 *flip-current-player*

```
flip_current_player: Player -> Player
flip_current_player(p) == if (p = <P1>) then <P2> else <P1>
post p <> RESULT;
```

definition
 flip-current-player :: *Player* $\Rightarrow$ *Player*
where
 flip-current-player p $\equiv$ (*if* (*p* = *P1*) *then P2 else P1*)

5.12.1 Specification

definition
 post-flip-current-player :: *Player* $\Rightarrow$ *Player* $\Rightarrow$ $\mathbb{B}$
where
 post-flip-current-player p RESULT $\equiv$ *p* $\neq$ *RESULT*

5.12.2 Satisfiability PO

definition
 PO-flip-current-player-sat-obl :: $\mathbb{B}$
where
 PO-flip-current-player-sat-obl $\equiv$
  $\forall$ *p* . ($\exists$ *r* . *post-flip-current-player p r*)

theorem *PO-flip-current-player-sat-obl*
unfolding *PO-flip-current-player-sat-obl-def post-flip-current-player-def*
by (*metis Player.distinct*(*1*))— SH

definition
 PO-flip-current-player-sat-exp-obl :: $\mathbb{B}$
where
 PO-flip-current-player-sat-exp-obl $\equiv$
  $\forall$ *p* . *post-flip-current-player p* (*flip-current-player p*)

theorem *PO-flip-current-player-sat-exp-obl*
unfolding *PO-flip-current-player-sat-exp-obl-def post-flip-current-player-def*
  flip-current-player-def
by *simp* — SH

6 VDM state

```
state Nim of
  limit: Move
  current: Player
  moves: Moves
inv mk_Nim(limit, current, moves) ==
  -- cannot move all at once
```

```
  limit < MAX_PILE
  and
  -- fair play
  fair_play(current, moves)
  and
  isFirst(<P1>)
--init nim == nim = mk_Nim(MAX_MOV, first_player(),
    FIXED_PLAY_GAME)
init nim == nim = mk_Nim(MAX_MOV, first_player(), [])
end
```

We use records to represent the VDM state. You can also use cartesian product or tuples. You need to represent the state invariant, its initialisation, and the result of the invariant on the given initial values.

record *NimSt* =
limit :: *Move*
current :: *Player*
moves :: *Moves*

VDM records field access ($x.moves$) is defined in Isabelle through functions (*moves x*), whereas record constants ($mkNimSt(l, c, m)$) are defined in Isabelle as (|*limit* = *l*, *current* = *c*, *moves* = *m*|). So, for instance, the result of

moves (|*limit* = *MAX-MOV*, *current* = *P1*, *moves* = [*1*, *2*]|)

is the sequence [*1*, *2*].

6.1 State invariant

For the state invariant we define a curried function with its components, checking the appropriate types first, and next the state invariant itself. Note that if the invariant makes use of auxiliary function definitions, it is implicitly adhering to those functions specifications as well (e.g. pre/post for *isFirst* and *fair-play*). Finally, we also define a version of the invariant on the state record itself.

definition
inv-Nim-flat :: *Move* ⇒ *Player* ⇒ *Moves* ⇒ $\mathbb{B}$
where
inv-Nim-flat l c ms ≡
inv-Move l ∧ *inv-Moves ms* ∧ *pre-fair-play c ms* ∧
post-fair-play c ms (*fair-play c ms*) ∧
l < *MAX-PILE* ∧ *fair-play c ms* ∧ *isFirst P1*

definition
inv-Nim :: *NimSt* ⇒ $\mathbb{B}$
where
inv-Nim st ≡ *inv-Nim-flat* (*limit st*) (*current st*) (*moves st*)

6.2 State initialisation

Initialisation is defined with an Isabelle record value. This of course must enforce the invariant as its postcondition.

definition
 init-Nim :: *NimSt*
where
 $init\text{-}Nim \equiv (\!|\ limit = MAX\text{-}MOV,\ current = P1,\ moves = [\,]\ |\!)$

definition
 post-init-Nim :: $\mathbb{B}$
where
 post-init-Nim $\equiv$ *inv-Nim init-Nim*

6.3 State satisfiability PO

definition
 PO-Nim-initialise-sat-obl :: $\mathbb{B}$
where
 PO-Nim-initialise-sat-obl $\equiv$ *True*

7 VDM operations

VDM operations, in so far as Isabelle is concerned, only require pre/post. That is because these are the parts that appear in the the proof obligations to be discharged. You might also want to define the explicit definition (e.g. the how), but is not strictly necessary. Explicit definitions are helpful. On the other hand, explicit witnesses for existential quantifiers, as discussed above for *best-move*, could lead to unprovable goals.

Preconditions depend on inputs and before state, whereas postconditions depend on inputs, outputs, before and after states in that order. Thus the boolean-valued function signature needs to be defined accordingly. Note that you need to check type invariants, as well as auxiliary function pre/post conditions on the appropriate arguments. For instance, *post-naive-choose-move* below references to *moves-left ms* is referring to the VDM after state (*moves ast*) of *Moves*.

7.1 *naive-choose-move* operation

```
naive_choose_move() r: Move ==
  -- naive choice: from 1 up to MAX_MOV or else amount left,
      presuming there are at least the last
  let m in set {1,...,min(MAX_MOV, moves_left(moves))} in return m
ext rd moves
pre moves_left(moves) > 0
post
```

```
-- might be = in the case of the loosing play
r <= moves_left(moves);
```

7.1.1 Specification

Notice that *moves-left* in the postcondition is applied to the after state (e.g. *moves ast*).

definition
 $pre\text{-}naive\text{-}choose\text{-}move :: NimSt \Rightarrow \mathbb{B}$
where
 $pre\text{-}naive\text{-}choose\text{-}move\ bst \equiv$
  $inv\text{-}Nim\ bst \wedge$
  $(let\ ms = (moves\ bst)\ in$
   $pre\text{-}moves\text{-}left\ ms \wedge$
   $post\text{-}moves\text{-}left\ ms\ (moves\text{-}left\ ms) \wedge$
   $\{1\ ..\ (min\ MAX\text{-}MOV\ (moves\text{-}left\ ms))\} \neq \{\} \wedge$
   $moves\text{-}left\ ms > 0)$

definition
 $post\text{-}naive\text{-}choose\text{-}move :: Move \Rightarrow NimSt \Rightarrow NimSt \Rightarrow \mathbb{B}$
where
 $post\text{-}naive\text{-}choose\text{-}move\ RESULT\ bst\ ast \equiv$
  $inv\text{-}Move\ RESULT \wedge inv\text{-}Nim\ bst \wedge inv\text{-}Nim\ ast \wedge$
  $(let\ ms = (moves\ ast)\ in$
   $pre\text{-}moves\text{-}left\ ms \wedge$
   $post\text{-}moves\text{-}left\ ms\ (moves\text{-}left\ ms) \wedge$
   $RESULT \leq moves\text{-}left\ (moves\ ast))$

7.1.2 Implementation

The implementation uses VDM's non deterministic (Hilbert's-)choice over a set. It can be encoded with Isabelle's Hilbert's choice operator[2]. Like in VDM, this has the precondition that the underlying set/sequence are not empty.

value $SOME\ m\ .\ m \in \{1\ ..\ MAX\text{-}MOV\}$
value $SOME\ m\ .\ m \in \{1\ ..\ (3::int)\}$

Operations should always return the sate and its result type. You could choose to avoid returning the state if there are no `ext wr` clauses declared (*i.e.* the operation is read-only and doesn't change the sate). This simplification is useful to avoid needing to handle tuples in proofs. I provide both versions for illustrative purposes.

definition
 $naive\text{-}choose\text{-}move0 :: NimSt \Rightarrow NimSt \times Move$
where

[2]See https://en.wikipedia.org/wiki/Choice_function

naive-choose-move0 st ≡
  (*st*, (*SOME m . m* ∈ {*1* .. (*min MAX-MOV* (*moves-left* (*moves st*)))}))

definition
  naive-choose-move :: *NimSt* ⇒ *Move*
where
  naive-choose-move st ≡
  (*SOME m . m* ∈ {*1* .. (*min MAX-MOV* (*moves-left* (*moves st*)))})

7.2 *first-player-winning-choose-move* operation

```
first_player_winning_choose_move() r: Move ==
  -- winning choice: get the best move, unless it's zero, so
      choose the least worst (1) play
  return max(1, best_move(moves))
ext rd moves, current
pre moves_left(moves) > 0
post
  -- can never be = moves_left(moves) or it would entail loosing?
  r < moves_left(moves)
  and
  -- after playing the chosen move r, the next player has no good
      move choice
  will_first_player_win() => best_move(play_move(current, r, moves
      )) = 0
  ;
```

7.2.1 Specification

definition
  pre-first-player-winning-choose-move :: *NimSt* ⇒ $\mathbb{B}$
where
  pre-first-player-winning-choose-move bst ≡
    inv-Nim bst ∧ *pre-moves-left* (*moves bst*) ∧ *moves-left* (*moves bst*) > *0*

definition
  post-first-player-winning-choose-move :: *Move* ⇒ *NimSt* ⇒ *NimSt* ⇒ $\mathbb{B}$
where
  post-first-player-winning-choose-move RESULT bst ast ≡
    inv-Move RESULT ∧ *inv-Nim bst* ∧ *inv-Nim ast* ∧
    (*let bms* = (*moves bst*) *in*
     let ams = (*moves ast*) *in*
      pre-moves-left ams ∧ *post-moves-left ams* (*moves-left ams*) ∧
      pre-who-plays-next ams ∧ *post-who-plays-next ams* (*who-plays-next ams*) ∧
      pre-will-first-player-win ∧
      (*let ac* = (*current ast*) *in*
       let pm = *play-move ac RESULT ams in*
       let bm = *best-move pm in*

$pre\text{-}play\text{-}move\ ac\ RESULT\ ams \wedge$
$post\text{-}play\text{-}move\ ac\ RESULT\ ams\ pm \wedge$
$pre\text{-}best\text{-}move\ pm \wedge post\text{-}best\text{-}move\ pm\ (best\text{-}move\ pm) \wedge$

$RESULT < moves\text{-}left\ ams \wedge$
$will\text{-}first\text{-}player\text{-}win \wedge$
$(isFirst\ (who\text{-}plays\text{-}next\ ams) \longrightarrow best\text{-}move\ pm = 0)$
$))$

7.2.2 Implementation

definition
first-player-winning-choose-move :: *NimSt* ⇒ *Move*
where
$first\text{-}player\text{-}winning\text{-}choose\text{-}move\ st \equiv max\ 1\ (best\text{-}move\ (moves\ st))$

7.2.3 Example PO: operation satisfiability

The satisfiability proof obligation of an operation Op under state St is:

$\forall\, input \in Type.$
$\quad \forall\, bst \in State.$
$\quad\quad pre\text{-}Op\ input\ bst \longrightarrow$
$\quad\quad (\exists\, output \in Type.\ \exists\, ast \in State.\ post\text{-}Op\ input\ output\ bst\ ast)$

That is, given any input and before state satisfying their invariants, if the precondition holds, then find witnesses for the output and after state, such that the postcondition holds. Operations without inputs or outputs can be declared similarly without the parameters. Operations with explicit definition have the witness choice fixed for the existential quantifiers.

Overture PO generator (POG) produces different versions of the satisfiability PO, depending on the kind of VDM declaration used (*e.g.* implicit, explicit, extended). In essence, the POG expand/simplifies definitions, as well as take advantage of explicit specification statements as witnesses to existential quantifiers. In doubt, use the general template above.

definition
PO-first-player-winning-choose-move-sat-obl :: $\mathbb{B}$
where
$PO\text{-}first\text{-}player\text{-}winning\text{-}choose\text{-}move\text{-}sat\text{-}obl \equiv$
$\quad \forall\ bst\ .\ pre\text{-}first\text{-}player\text{-}winning\text{-}choose\text{-}move\ bst \longrightarrow$
$\quad\quad (\exists\ RESULT\ ast\ .\ post\text{-}first\text{-}player\text{-}winning\text{-}choose\text{-}move\ RESULT\ bst\ ast)$

definition
PO-first-player-winning-choose-move-sat-exp-obl :: $\mathbb{B}$
where
$PO\text{-}first\text{-}player\text{-}winning\text{-}choose\text{-}move\text{-}sat\text{-}exp\text{-}obl \equiv$
$\quad \forall\ bst\ .\ pre\text{-}first\text{-}player\text{-}winning\text{-}choose\text{-}move\ bst \longrightarrow$

post-first-player-winning-choose-move (*max 1* (*best-move* (*moves bst*))) *bst bst*

As an illustration, a naive attempt at these kind of proofs by simply expanding definitions and doing layered simplificaiton will only work if appropriate lemmas are in place. Previous proofs of satisfiability of involved functions will also be important in these POs about top-level operations.

theorem *PO-first-player-winning-choose-move-sat-exp-obl*
unfolding *PO-first-player-winning-choose-move-sat-exp-obl-def*
unfolding *pre-first-player-winning-choose-move-def*
post-first-player-winning-choose-move-def Let-def
apply *auto*
oops

7.3 *fixed-choose-move* operation

```
fixed_choose_move() r: Move ==
  return FIXED_PLAY(len moves + 1)
ext rd moves, current
pre moves_left(moves) > 0
post
  -- can never be = moves_left(moves) or it would entail loosing?
  r < moves_left(moves)
  and
  -- after playing the chosen move r, the next player has no good
      move choice
  (will_first_player_win()
  and
  isFirst(who_plays_next(moves)))
  => best_move(play_move(current, r, moves)) = 0
  ;

values
                  -- 1 2 1 2 1 2 1 2 1 2
FIXED_PLAY: Moves = [3,2,2,1,3,2,2,1,3,1];
```

The *FIXED-PLAY* value needs to be declared first as it is used in the coming definition. Also, it needs to satisfy the invariant of *Moves* in the precondition of where it appears. We use *definition* instead of *abbreviation* to avoid expansion in proofs.

definition
FIXED-PLAY :: *Moves*
where
FIXED-PLAY ≡ [*3*,*2*,*2*,*1*,*3*,*2*,*2*,*1*,*3*,*1*]

definition
inv-FIXED-PLAY :: $\mathbb{B}$
where
inv-FIXED-PLAY ≡ *inv-Moves FIXED-PLAY*

7.3.1 Specification

definition
pre-fixed-choose-move :: *NimSt* $\Rightarrow$ $\mathbb{B}$
where
pre-fixed-choose-move bst $\equiv$
pre-first-player-winning-choose-move bst

definition
post-fixed-choose-move :: *Move* $\Rightarrow$ *NimSt* $\Rightarrow$ *NimSt* $\Rightarrow$ $\mathbb{B}$
where
post-fixed-choose-move RESULT bst ast $\equiv$
post-first-player-winning-choose-move RESULT bst ast

7.3.2 Implementation

definition
fixed-choose-move :: *NimSt* $\Rightarrow$ *Move*
where
fixed-choose-move st $\equiv$ *applyVDMSeq FIXED-PLAY* $(len\ (moves\ st) + 1)$

7.4 *save* operation

```
save(choice : Move) ==
  (dcl ms  : Moves := play_move(current, choice, moves),
       next: Player := flip_current_player(current);
    --flip_player();, see flip_current_player(current) instead
    -- to keep the fair_play_invariant, we need to change both
        atomically
    atomic(
      moves := ms;
      current := next;
    );
    -- we want to debug who played last, so flip back
    debug(flip_current_player(current), choice);
  )
ext wr current, moves
pre pre_play_move(current, choice, moves)
post
  post_play_move(current, choice, moves~, moves)
  and
  current~ <> current
```

7.4.1 Specification

definition
pre-save :: *Move* $\Rightarrow$ *NimSt* $\Rightarrow$ $\mathbb{B}$
where

pre-save choice bst $\equiv$
inv-Nim bst $\wedge$ *inv-Move choice* $\wedge$
$($*let bc* $=$ $($*current bst*$)$ *in*
let bms$=$ $($*moves bst*$)$ *in*
pre-play-move bc choice bms $\wedge$
post-play-move bc choice bms
$($*play-move bc choice bms*$))$

definition
post-save :: *Move* $\Rightarrow$ *NimSt* $\Rightarrow$ *NimSt* $\Rightarrow$ $\mathbb{B}$
where
post-save choice bst ast $\equiv$
inv-Nim bst $\wedge$ *inv-Nim ast* $\wedge$ *inv-Move choice* $\wedge$
$($*let bc* $=$ $($*current bst*$)$ *in*
let ac $=$ $($*current ast*$)$ *in*
let bms$=$ $($*moves bst*$)$ *in*
let ams$=$ $($*moves ast*$)$ *in*
pre-play-move bc choice bms $\wedge$
post-play-move bc choice bms $($*play-move bc choice bms*$)$ $\wedge$
post-flip-current-player bc $($*flip-current-player bc*$)$ $\wedge$

pre-play-move ac choice ams $\wedge$
post-play-move ac choice bms ams
$)$

7.4.2 Implementation

For read-write operations, the after state must be returned together with any result value as a tupple or extended record. Like with read-only operations, if result is void, then just the state is enough as a result type to avoid needing to handle tuples unnecessarily.

Local variable declarations can be translated using *Let* expressions. Because Isabelle is always functional (*i.e.* referencially transparent), there is no need for atomic statements (*i.e.* there aren't any state updates as such: a new state is built and returned as a result). You can either rebuild the whole state as a new record (*save*) or use record update syntax (*save2*).

definition
save :: *Move* $\Rightarrow$ *NimSt* $\Rightarrow$ *NimSt*
where
save choice bst $\equiv$
$($*let ms* $=$ *play-move* $($*current bst*$)$ *choice* $($*moves bst*$)$;
next $=$ *flip-current-player* $($*current bst*$)$ *in*
$(\!|$ *limit* $=$ $($*limit bst*$)$, *current* $=$ *next*, *moves* $=$ *ms* $|\!))$

definition
save0 :: *NimSt* $\Rightarrow$ *Move* $\Rightarrow$ *NimSt*
where

$save0\ bst\ choice \equiv$
$(let\ ms = play\text{-}move\ (current\ bst)\ choice\ (moves\ bst);$
$next = flip\text{-}current\text{-}player\ (current\ bst)\ in$
$bst\ (\!|\ current := next, moves := ms\ |\!))$

7.5 *who-won* operation

```
-- who won is determined by who played more moves?
who_won() w: Player ==
  return current -- who_plays_next(moves)
ext rd current, moves
pre isFirst(first_player())
post (who_won_invariant(w, moves)
      and
      -- last save flipped loser and put winner as current
      w = current)
```

7.5.1 Specification

definition
$pre\text{-}who\text{-}won :: NimSt \Rightarrow \mathbb{B}$
where
$pre\text{-}who\text{-}won\ bst = inv\text{-}Nim\ bst$

definition
$post\text{-}who\text{-}won :: Player \Rightarrow NimSt \Rightarrow NimSt \Rightarrow \mathbb{B}$
where
$post\text{-}who\text{-}won\ RESULT\ bst\ ast \equiv$
$inv\text{-}Nim\ bst \wedge inv\text{-}Nim\ ast \wedge$
$(let\ ams = (moves\ ast)\ in$
$pre\text{-}who\text{-}won\text{-}invariant\ RESULT\ ams \wedge$
$post\text{-}who\text{-}won\text{-}invariant\ RESULT\ ams\ (who\text{-}won\text{-}invariant\ RESULT\ ams))$

7.5.2 Implementation

definition
$who\text{-}won :: NimSt \Rightarrow Player$
where
$who\text{-}won\ bst \equiv (current\ bst)$

7.6 *tally* operation

```
tally() ==
   (print("\nPlayer ");print(who_won());println(" won! Play
      finished with:");
```

```
    print("\tP1 moves = ");println(moves_of(moves, isFirst(<P1>)))
        ;
    print("\tP2 moves = ");println(moves_of(moves, isFirst(<P2>)))
        ;
   )
ext rd current, moves;
```

7.6.1 Specification

definition
pre-tally :: *NimSt* $\Rightarrow \mathbb{B}$
where
pre-tally bst $\equiv$ *inv-Nim bst* $\wedge$ *pre-who-won bst*

definition
post-tally :: *NimSt* $\Rightarrow$ *NimSt* $\Rightarrow \mathbb{B}$
where
post-tally bst ast $\equiv$
inv-Nim bst $\wedge$ *inv-Nim ast* $\wedge$ *pre-who-won bst* $\wedge$
post-who-won (*current ast*) *bst ast*

7.6.2 Implementation

We define tally in VDM to illustrate the use of sequential composition. We will not show I/O in Isabelle.

definition
tally :: *NimSt* $\Rightarrow$ *NimSt*
where
tally bst $\equiv$ (*let p* $=$ *who-won bst in bst*)

7.7 VDM while statement in Isabelle

The VDM while statement

```
  (while b do c) s
```

where s is the before state that both the loop condition b and the loop body c can talk about, can be translated to Isabelle using the *while* combinator as

while (λ *st* . *b*)
(λ *st* . *c*)
bst

while is defined in terms of a boolean-valued function from the state for the loop condition, a homogeneous function from the state for the loop body, and the initial state itself. Sequential composition can be achieved with functional composition. For example the VDM statement

```
(f(in) ; g(in))
```

where (in, st) are the inputs and (implicit) before state, can be translated to Isabelle as $(g\ in\ (f\ in\ s))$. That is, the before state of g is the after state of f executing on the given input and before state.

As loops operate on intermediate values, they have different specification conditions as the pre/post of operation's at entry/exit points. To ensure that type invariant consistency, as well as auxiliary functions and operations pre/post conditions are enforced, we create auxiliary Isabelle definitions to enable us to call the appropriate pre/post at the right place. Moreover, loops should contain an invariant and variant statement (**TODO**!).

7.8 *naive-play-game* operation

```
naive_play_game() ==
  ((while moves_left(moves) > 0 do
      save(naive_choose_move())
    );
   tally()
  )
ext wr current, moves
pre moves_left(moves) = MAX_PILE
post moves_left(moves) = 0;
```

7.8.1 Specification

definition
 pre-naive-play-game :: *NimSt* ⇒ $\mathbb{B}$
where
 pre-naive-play-game bst ≡
  inv-Nim bst ∧
  (*let bms* = (*moves bst*) *in*
   pre-moves-left bms ∧
   post-moves-left bms (*moves-left bms*) ∧
   moves-left bms = *MAX-PILE*)

definition
 post-naive-play-game :: *NimSt* ⇒ *NimSt* ⇒ $\mathbb{B}$
where
 post-naive-play-game bst ast ≡
  inv-Nim bst ∧ *inv-Nim ast* ∧

$(let\ ams = (moves\ ast)\ in$
$pre\text{-}moves\text{-}left\ ams \wedge$
$post\text{-}moves\text{-}left\ ams\ (moves\text{-}left\ ams) \wedge$
$moves\text{-}left\ ams = 0)$

7.8.2 Implementation

definition
naive-play-game-inner-play :: *NimSt* ⇒ *NimSt*
where
naive-play-game-inner-play bst ≡
save (*naive-choose-move bst*) *bst*

definition
naive-play-game-loop :: *NimSt* ⇒ *NimSt*
where
naive-play-game-loop bst ≡
while (λ *bst* . *moves-left* (*moves bst*) > *0*)
(λ *bst* . *save* (*naive-choose-move bst*) *bst*)
bst

definition
naive-play-game :: *NimSt* ⇒ *NimSt*
where
naive-play-game bst ≡
tally (*naive-play-game-loop bst*)

7.9 *fixed-play-game* operation

```
fixed_play_game() ==
  ((while moves_left(moves) > 0 do
      save(fixed_choose_move())
    );
   tally()
  )
ext wr current, moves
pre moves_left(moves) = MAX_PILE
post moves_left(moves) = 0;
```

7.9.1 Specification

TODO

definition
pre-fixed-play-game :: *NimSt* ⇒ $\mathbb{B}$
where

pre-fixed-play-game bst ≡ *True*

definition
post-fixed-play-game :: *NimSt* ⇒ *NimSt* ⇒ $\mathbb{B}$
where
post-fixed-play-game bst ast ≡ *True*

7.9.2 Implementation

definition
fixed-play-game-loop :: *NimSt* ⇒ *NimSt*
where
fixed-play-game-loop bst ≡
  while (λ *bst* . *moves-left* (*moves bst*) > *0*)
    (λ *bst* . *save* (*fixed-choose-move bst*) *bst*)
    bst

definition
fixed-play-game :: *NimSt* ⇒ *NimSt*
where
fixed-play-game bst ≡ *tally* (*fixed-play-game-loop bst*)

7.10 *first-win-game* operation

```
first_win_game() ==
  ((while moves_left(moves) > 0 do
    (dcl choice : Move := (if (isFirst(current)) then
                             first_player_winning_choose_move()
                           else
                             naive_choose_move());
      save(choice)
    )
   );
   tally()
  )
ext wr current, moves
pre moves_left(moves) = MAX_PILE
post moves_left(moves) = 0;
```

7.10.1 Specification

TODO?

definition
pre-first-win-game :: *NimSt* ⇒ $\mathbb{B}$
where
pre-first-win-game bst ≡ *pre-naive-play-game bst*

definition
post-first-win-game :: *NimSt* ⇒ *NimSt* ⇒ $\mathbb{B}$
where
post-first-win-game bst ast ≡ *post-naive-play-game bst ast*

7.10.2 Implementation

definition
first-win-game-loop :: *NimSt* ⇒ *NimSt*
where

```
first-win-game-loop bst ≡
  while (λ bst . moves-left (moves bst) > 0)
     (λ bst . (let choice = (if (isFirst (current bst)) then
                          first-player-winning-choose-move bst
                        else
                          naive-choose-move bst
                        )
              in (save choice bst)
             )
     ) bst
```

definition
first-win-game :: *NimSt* ⇒ *NimSt*
where

```
first-win-game bst ≡
  tally (first-win-game-loop bst)
```

8 VDM proof obligations

The Overture proof obligation generator (POG) can be executed either from the context menu of the corresponding project, via the command line, or via the console. The context menu fills in the PO explorer view, whereas the console prints POs in Overture/VDM syntax. If you run the console, it is easier to copy-and-paste the POs' text for translation to Isabelle. The console POG will generate POs for all modules in the project. You should be careful to only consider the POs from modules of interest only. To avoid confusion, PO names should be like their corresponding description prefixed with PO, and be declared as a $\mathbb{B}$ definition to be proved. Note that Isabelle will not declare implicitly enforced/expected type invariants. So just like for other definitions, type invariants need to be explicit added for quantified variables. Isabelle on the other hand, will do base type inference. For `NimFull.vdmsl`, POG generated 40 POs, some of which I discuss below.

8.1 PO1

Move: type invariant satisfiable obligation @ in 'NimFull' (./NimFull.vdmsl) at line 23:1

```
(exists m:Move & (m <= MAX_MOV))
Proof Obligation 01: (Unproved)
```

definition
PO01-move-type-inv-sat-obl :: $\mathbb{B}$
where
PO01-move-type-inv-sat-obl $\equiv \exists$ *m* . *inv-Move m* $\wedge$ *m* $\leq$ *MAX-MOV*

theorem *PO01-move-type-inv-sat-obl*
unfolding *PO01-move-type-inv-sat-obl-def inv-Move-def*
using *inv-VDMNat1-def* **by** *force* — SH

definition
PO01-move-type-inv-sat-obl-gen :: $\mathbb{B}$
where
PO01-move-type-inv-sat-obl-gen $\equiv \exists$ *m* . *inv-VDMNat1 m* $\wedge$ *m* $\leq$ *G-MAX-MOV* $\wedge$ *m* $\leq$ *G-MAX-MOV*

theorem *PO01-move-type-inv-sat-obl-gen*
unfolding *PO01-move-type-inv-sat-obl-gen-def*
using *inv-VDMNat1-def n1-MM* **by** *blast* — SH

8.2 PO2

```
Moves: legal sequence application obligation @ in 'NimFull' (./
    NimFull.vdmsl) at line 32:30
(forall s:seq of (Move) & ((sum_elems(s) <= MAX_PILE) => ((
    sum_elems(s) = MAX_PILE) => ((len s) in set (inds s)))))
Proof Obligation 02: (Unproved)
```

For universally quantified proofs, type invariants are to be considered as a guard. That is, if the invariant hold, then the PO must follow; otherwise, we do not care. That is an accurate representation for what Isabelle type inference does to the bound variables. For instance

$$\forall x{\in}\mathbb{N}.\ 0 < f\, x = \forall x.\ x \in \mathbb{N} \longrightarrow 0 < f\, x$$

So, whenever $x \notin \mathbb{N}$, then we do not care about the value of the expression.

value *len* [*a*,*b*]
value *inds* [*a*,*b*]

definition
PO02-moves-legal-seq-app-obl :: $\mathbb{B}$
where

PO02-moves-legal-seq-app-obl $\equiv \forall\ s\ .\ (inv\text{-}SeqElems\ inv\text{-}Move\ s) \longrightarrow$
$(sum\text{-}elems\ s \leq MAX\text{-}PILE \longrightarrow (sum\text{-}elems\ s = MAX\text{-}PILE) \longrightarrow (len\ s \in inds\ s))$

theorem *PO02-moves-legal-seq-app-obl*
unfolding *PO02-moves-legal-seq-app-obl-def inv-Moves-def VDMSeq-def*
apply (*intro allI impI*)
apply *simp*
apply (*erule sum-elems.elims*)
apply *simp+*
done

8.3 PO3

```
Moves: type invariant satisfiable obligation @ in 'NimFull' (./
    NimFull.vdmsl) at line 26:1
(exists s:Moves & ((sum_elems(s) <= MAX_PILE) and ((sum_elems(s) =
     MAX_PILE) => (s((len s)) = 1))))
Proof Obligation 03: (Unproved)
```

For commonly used combinations of definitions to be unfolded, you can use a *lemmas* command to give a synonym for a group of definitions.

definition
PO03-moves-type-inv-sat-obl :: $\mathbb{B}$
where
PO03-moves-type-inv-sat-obl $\equiv \exists\ s\ .\ inv\text{-}Moves\ s \wedge$
$(sum\text{-}elems\ s \leq MAX\text{-}PILE \longrightarrow (sum\text{-}elems\ s = MAX\text{-}PILE) \longrightarrow applyVDMSeq\ s\ (len\ s) = 1)$

theorem *PO03-moves-type-inv-sat-obl*
unfolding *PO03-moves-type-inv-sat-obl-def VDMSeq-def* **oops**

Postcondition of *sum-elems* is just *True*, hence this

8.4 PO4

```
sum_elems: function establishes postcondition obligation @ in '
    NimFull' (./NimFull.vdmsl) at line 37:1
(forall s:seq of (Move) & post_sum_elems(s, (cases s :
[] -> 0,
[x] ^ xs -> (x + sum_elems(xs))
 end)))
Proof Obligation 04: (Unproved)
```

definition
PO04-sum-elems-post-obl :: $\mathbb{B}$

where
PO04-sum-elems-post-obl $\equiv \forall$ *ms . inv-SeqElems inv-Move ms* $\longrightarrow$
post-sum-elems ms (*case ms of* [] $\Rightarrow$ *0* | (*x#xs*) $\Rightarrow$ *x* + *sum-elems xs*)

theorem *PO04-sum-elems-post-obl*
unfolding *PO04-sum-elems-post-obl-def inv-Move-def pre-sum-elems-def post-sum-elems-def*
apply (*rule allI*)
apply (*case-tac ms*)
apply (*intro impI conjI iffI*,*simp-all*)
apply (*frule l-sum-elems-nat*)
apply *simp*
apply (*subgoal-tac inv-SeqElems inv-Move ms*)
apply (*frule l-sum-elems-nat*)
apply (*simp add*: *l-sum-elems-nat*)
find-theorems *sum-elems* -
oops

Because *sum-elems* is recursively defined in Isabelle, its proof obligations from Overure related to recursive definitions are irrelevant. That is because Isabelle automatically proves such POs implicitly. For example,

$[\![?P\ []; \bigwedge x\ xs.\ ?P\ xs \Longrightarrow ?P\ (x \# xs)]\!] \Longrightarrow ?P\ ?a0.0$
$[\![?x = [] \Longrightarrow ?P; \bigwedge x\ xs.\ ?x = x \# xs \Longrightarrow ?P]\!] \Longrightarrow ?P$

theory *NimFullProofs*
imports *NimFull*
begin

9 Role of lemmas

Some lemmas proved in the process of discovering the proofs, a few turned out not to be necessary in the final proof, but helped in discovering the problems with the precondition of *play-move*.

9.1 Satisfiability PO of *play-move*

definition
PO-play-move-sat-obl0 :: $\mathbb{B}$
where
PO-play-move-sat-obl0 $\equiv \forall$ *p m s . inv-Move m* $\longrightarrow$ *inv-Moves s* $\longrightarrow$
pre-play-move0 p m s $\longrightarrow$ ($\exists$ *r . post-play-move p m s r*)

theorem *PO-play-move-sat-obl0*
unfolding *PO-play-move-sat-obl0-def pre-play-move0-def post-play-move-def*
apply *simp*
apply (*intro allI impI conjI*,*elim conjE*)
apply (*rule-tac x*=*s* @ [*m*] **in** *exI*, *simp*)

$$
\begin{array}{l}
\mathit{1.}\ \bigwedge p\ m\ s.\\
\quad [\![\mathit{inv\text{-}Move}\ m;\ \mathit{inv\text{-}Moves}\ s;\ \mathit{pre\text{-}moves\text{-}left}\ s;\ \mathit{pre\text{-}fair\text{-}play}\ p\ s;\\
\quad \mathit{post\text{-}fair\text{-}play}\ p\ s\ \mathit{True};\ \mathit{moves\text{-}left}\ s \neq \mathit{1} \longrightarrow m < \mathit{moves\text{-}left}\ s;\\
\quad \mathit{0} < \mathit{moves\text{-}left}\ s;\ \mathit{fair\text{-}play}\ p\ s]\!]\\
\quad \Longrightarrow \mathit{inv\text{-}Moves}\ (s\ @\ [m]) \wedge\\
\qquad \mathit{pre\text{-}sum\text{-}elems}\ s \wedge\\
\qquad \mathit{pre\text{-}sum\text{-}elems}\ (s\ @\ [m]) \wedge\\
\qquad \mathit{post\text{-}sum\text{-}elems}\ s\ (\mathit{sum\text{-}elems}\ s) \wedge\\
\qquad \mathit{post\text{-}sum\text{-}elems}\ (s\ @\ [m])\ (\mathit{sum\text{-}elems}\ (s\ @\ [m])) \wedge\\
\qquad \mathit{sum\text{-}elems}\ s < \mathit{sum\text{-}elems}\ (s\ @\ [m]) \wedge\\
\qquad \mathit{sum\text{-}elems}\ s + m = \mathit{sum\text{-}elems}\ (s\ @\ [m])
\end{array}
$$

These goals will require various lemmas.

oops

definition
PO-play-move-sat-exp-obl0 :: $\mathbb{B}$
where
$\mathit{PO\text{-}play\text{-}move\text{-}sat\text{-}exp\text{-}obl0} \equiv \forall\ p\ m\ s\ .\ \mathit{inv\text{-}Move}\ m \longrightarrow \mathit{inv\text{-}Moves}\ s \longrightarrow$
$\quad \mathit{pre\text{-}play\text{-}move0}\ p\ m\ s \longrightarrow \mathit{post\text{-}play\text{-}move}\ p\ m\ s\ (\mathit{play\text{-}move}\ p\ m\ s)$

9.2 Lemmas about auxiliary function *sum-elems*

fun *nconcat* :: $\mathbb{Z}$ *list* $\Rightarrow$ $\mathbb{Z}$ *list* $\Rightarrow$ $\mathbb{Z}$ *list*
where
$\mathit{nconcat}\ []\ \mathit{ys} \quad = \mathit{ys}$
$|\ \mathit{nconcat}\ (x\#\mathit{xs})\ \mathit{ys} = x\ \#\ (\mathit{nconcat}\ \mathit{xs}\ \mathit{ys})$

lemma *l-concat-append* : $\mathit{nconcat}\ \mathit{xs}\ \mathit{ys} = \mathit{xs}\ @\ \mathit{ys}$
apply (*induct ys*, *simp-all*) **oops**

lemma *l-concat-append* : $\mathit{nconcat}\ \mathit{xs}\ \mathit{ys} = \mathit{xs}\ @\ \mathit{ys}$
by (*induct xs*, *simp-all*)

lemma *l-sum-elems-nconcat*: $\mathit{sum\text{-}elems}\ (\mathit{nconcat}\ \mathit{ms}\ [m]) = (m + \mathit{sum\text{-}elems}\ \mathit{ms})$

apply (*induct ms*, *simp-all*) **done**

Some interesting lemmas about *sum-elems*

$\mathit{inv\text{-}SeqElems}\ \mathit{inv\text{-}Move}\ ?s \Longrightarrow \mathit{0} \leq \mathit{sum\text{-}elems}\ ?s$
$[\![\mathit{inv\text{-}SeqElems}\ \mathit{inv\text{-}Move}\ ?s;\ ?s \neq []]\!] \Longrightarrow \mathit{0} < \mathit{sum\text{-}elems}\ ?s$
$\mathit{inv\text{-}SeqElems}\ \mathit{inv\text{-}Move}\ ?s \Longrightarrow (\mathit{0} < \mathit{sum\text{-}elems}\ ?s) = (?s \neq [])$

9.3 Lemma discovery through failed proof attempts

1 Naive attempt: layered expansion followed by simplification.

theorem *PO-play-move-sat-exp-obl0*
unfolding *PO-play-move-sat-exp-obl0-def post-play-move-def play-move-def*
apply (*intro allI impI conjI*)

1. $\bigwedge p\ m\ s.\ [\![\mathit{inv\text{-}Move}\ m;\ \mathit{inv\text{-}Moves}\ s;\ \mathit{pre\text{-}play\text{-}move0}\ p\ m\ s]\!] \Longrightarrow \mathit{inv\text{-}Move}\ m$
2. $\bigwedge p\ m\ s.\ [\![\mathit{inv\text{-}Move}\ m;\ \mathit{inv\text{-}Moves}\ s;\ \mathit{pre\text{-}play\text{-}move0}\ p\ m\ s]\!] \Longrightarrow \mathit{inv\text{-}Moves}\ s$
3. $\bigwedge p\ m\ s.$
   $[\![\mathit{inv\text{-}Move}\ m;\ \mathit{inv\text{-}Moves}\ s;\ \mathit{pre\text{-}play\text{-}move0}\ p\ m\ s]\!]$
   $\Longrightarrow \mathit{inv\text{-}Moves}\ (s\ @\ [m])$
4. $\bigwedge p\ m\ s.$
   $[\![\mathit{inv\text{-}Move}\ m;\ \mathit{inv\text{-}Moves}\ s;\ \mathit{pre\text{-}play\text{-}move0}\ p\ m\ s]\!] \Longrightarrow \mathit{pre\text{-}sum\text{-}elems}\ s$
5. $\bigwedge p\ m\ s.$
   $[\![\mathit{inv\text{-}Move}\ m;\ \mathit{inv\text{-}Moves}\ s;\ \mathit{pre\text{-}play\text{-}move0}\ p\ m\ s]\!]$
   $\Longrightarrow \mathit{pre\text{-}sum\text{-}elems}\ (s\ @\ [m])$
6. $\bigwedge p\ m\ s.$
   $[\![\mathit{inv\text{-}Move}\ m;\ \mathit{inv\text{-}Moves}\ s;\ \mathit{pre\text{-}play\text{-}move0}\ p\ m\ s]\!]$
   $\Longrightarrow \mathit{post\text{-}sum\text{-}elems}\ s\ (\mathit{sum\text{-}elems}\ s)$
7. $\bigwedge p\ m\ s.$
   $[\![\mathit{inv\text{-}Move}\ m;\ \mathit{inv\text{-}Moves}\ s;\ \mathit{pre\text{-}play\text{-}move0}\ p\ m\ s]\!]$
   $\Longrightarrow \mathit{post\text{-}sum\text{-}elems}\ (s\ @\ [m])\ (\mathit{sum\text{-}elems}\ (s\ @\ [m]))$
8. $\bigwedge p\ m\ s.$
   $[\![\mathit{inv\text{-}Move}\ m;\ \mathit{inv\text{-}Moves}\ s;\ \mathit{pre\text{-}play\text{-}move0}\ p\ m\ s]\!]$
   $\Longrightarrow \mathit{sum\text{-}elems}\ s < \mathit{sum\text{-}elems}\ (s\ @\ [m])$
9. $\bigwedge p\ m\ s.$
   $[\![\mathit{inv\text{-}Move}\ m;\ \mathit{inv\text{-}Moves}\ s;\ \mathit{pre\text{-}play\text{-}move0}\ p\ m\ s]\!]$
   $\Longrightarrow \mathit{sum\text{-}elems}\ s + m = \mathit{sum\text{-}elems}\ (s\ @\ [m])$

The subgoals come directly from the *post-play-move* for the given witness:

$\mathit{post\text{-}play\text{-}move}\ p\ m\ \mathit{ms}\ (\mathit{ms}\ @\ [m]) \equiv$
$\mathit{inv\text{-}Move}\ m \wedge$
$\mathit{inv\text{-}Moves}\ \mathit{ms} \wedge$
$\mathit{inv\text{-}Moves}\ (\mathit{ms}\ @\ [m]) \wedge$
$\mathit{pre\text{-}sum\text{-}elems}\ \mathit{ms} \wedge$
$\mathit{pre\text{-}sum\text{-}elems}\ (\mathit{ms}\ @\ [m]) \wedge$
$\mathit{post\text{-}sum\text{-}elems}\ \mathit{ms}\ (\mathit{sum\text{-}elems}\ \mathit{ms}) \wedge$
$\mathit{post\text{-}sum\text{-}elems}\ (\mathit{ms}\ @\ [m])\ (\mathit{sum\text{-}elems}\ (\mathit{ms}\ @\ [m])) \wedge$
$\mathit{sum\text{-}elems}\ \mathit{ms} < \mathit{sum\text{-}elems}\ (\mathit{ms}\ @\ [m]) \wedge$
$\mathit{sum\text{-}elems}\ \mathit{ms} + m = \mathit{sum\text{-}elems}\ (\mathit{ms}\ @\ [m])$

After simplifying the already parts of the input invariants, we get

apply (*simp-all*)

1. $\bigwedge p\ m\ s.$
   $[\![\mathit{inv\text{-}Move}\ m;\ \mathit{inv\text{-}Moves}\ s;\ \mathit{pre\text{-}play\text{-}move0}\ p\ m\ s]\!]$
   $\Longrightarrow \mathit{inv\text{-}Moves}\ (s\ @\ [m])$
2. $\bigwedge p\ m\ s.$

$[\![$*inv-Move m*; *inv-Moves s*; *pre-play-move0 p m s*$]\!] \Longrightarrow$ *pre-sum-elems s*

3. $\bigwedge$*p m s*.
   $[\![$*inv-Move m*; *inv-Moves s*; *pre-play-move0 p m s*$]\!]$
   $\Longrightarrow$ *pre-sum-elems* (*s* @ [*m*])
4. $\bigwedge$*p m s*.
   $[\![$*inv-Move m*; *inv-Moves s*; *pre-play-move0 p m s*$]\!]$
   $\Longrightarrow$ *post-sum-elems s* (*sum-elems s*)
5. $\bigwedge$*p m s*.
   $[\![$*inv-Move m*; *inv-Moves s*; *pre-play-move0 p m s*$]\!]$
   $\Longrightarrow$ *post-sum-elems* (*s* @ [*m*]) (*sum-elems* (*s* @ [*m*]))
6. $\bigwedge$*p m s*.
   $[\![$*inv-Move m*; *inv-Moves s*; *pre-play-move0 p m s*$]\!]$
   $\Longrightarrow$ *sum-elems s* $<$ *sum-elems* (*s* @ [*m*])
7. $\bigwedge$*p m s*.
   $[\![$*inv-Move m*; *inv-Moves s*; *pre-play-move0 p m s*$]\!]$
   $\Longrightarrow$ *sum-elems s* $+$ *m* $=$ *sum-elems* (*s* @ [*m*])

We will create a lemma for each expression that is not already part of the precondition. Moreover, it is interesting that *fair-play* does not appear in the post condition: it ought to.

I will tackle the expressions from simplest to most complex. This is a useful tactic as simpler goals will be easier to prove.

oops

9.3.1 Lemmas per subgoal

The precondition knows about *pre-moves-left*, which knows about *pre-sum-elems*. The next lemma weakens the goal: if you get a *pre-sum-elems* to handle, you can exchange it with a *pre-moves-left*. This fits with the necessary proof to do, but is not quite a general lemma.

lemma *l-moves-left-pre-sume*: *pre-moves-left ms* $\Longrightarrow$ *pre-sum-elems ms*
by (*simp add*: *pre-moves-left-def*) — SH, subgoal 2

lemma *l-pre-sume-seqelems-move*: *inv-SeqElems inv-Move ms* $\Longrightarrow$ *pre-sum-elems ms*
by (*simp add*: *pre-sum-elems-def*) — SH, subgoal 2

The next lemma helps Isabelle infer (forwardly) that, if *inv-Moves ms* holds, then so would the smaller claim that all elements within the sequence respect *inv-Move*. As you will see in proofs below, this lemma is useful in bridging the gap between what is needed for the lemma proof, and what is available in the goal where the lemma is to be used (i.e. the simpler the lemma conditions the better/most applicable the lemma will be).

lemma *l-inv-Moves-inv-SeqElems*: *inv-Moves ms* $\Longrightarrow$ *inv-SeqElems inv-Move ms*
using *inv-Moves-def* **by** *blast* — SH, useful for subgoal 2

lemma *l-sg2-pre-sume*: *inv-Moves ms* $\Longrightarrow$ *pre-sum-elems ms*
using *inv-Moves-def* **by** *blast* — SH, subgoal 2

These synonyms for lemmas/definition groups is useful not only to avoid long unfolding chains but also to help sledgehammer know bout related concepts.

lemma *l-sg3-pre-sume-append*: *inv-Move m* $\Longrightarrow$ *inv-Moves ms* $\Longrightarrow$ *pre-sum-elems* (*ms* @ [*m*])
oops

lemmas *inv-Move-defs* = *inv-Move-def inv-VDMNat1-def max-def*
lemmas *inv-Moves-defs* = *inv-Moves-def inv-SeqElems-def pre-sum-elems-def post-sum-elems-def*

lemma *l-sg3-pre-sume-append*: *inv-Move m* $\Longrightarrow$ *inv-Moves ms* $\Longrightarrow$ *pre-sum-elems* (*ms* @ [*m*])
unfolding *inv-Moves-defs play-move-def Let-def* **by** *simp* — SH, subgoal 3

lemma *l-sg4-post-sume*: *inv-SeqElems inv-Move ms* $\Longrightarrow$ *post-sum-elems ms* (*sum-elems ms*)
unfolding *post-sum-elems-def*
by (*simp add*: *inv-VDMNat-def l-pre-sum-elems l-sum-elems-nat*) — SH, subgoal 4

lemma *l-sg5-post-sume-append*: *inv-Move m* $\Longrightarrow$ *inv-Moves ms* $\Longrightarrow$ *post-sum-elems* (*ms* @ [*m*]) (*sum-elems* (*ms* @ [*m*]))
unfolding *post-sum-elems-def*
by (*metis l-inv-Moves-inv-SeqElems l-inv-SeqElems-append l-sg4-post-sume post-sum-elems-def*)
— SH, subgoal5

9.3.2 New (general) lemmas about *sum-elems*

The actual VDM (declared) postcondition represents subgoals 6 and 7. Those are discharged by the most general of lemmas here. It is a nice property of *sum-elems*: it distributes over concatenation and is exchanged for summation, on singleton lists as well as in general. It is often better to give general lemmas as they are more applicable, and surprisingly, easier to prove.

lemma *l-sum-elems-append*: *sum-elems* (*ms* @ [*m*]) = (*m* + *sum-elems ms*)
by (*induct ms*, *simp-all*)

lemma *l-sum-elems-append-gen*: *sum-elems* (*s* @ *t*) = (*sum-elems s* + *sum-elems t*)
by (*induct s*, *simp-all*)

9.4 "Sledgehammerable proofs"

2 Lemma-based attempt with sledgehammer support.

Let us see if our lemmas are working: will sledgehammer find the proofs?

theorem *PO-play-move-sat-exp-obl0*

...

1. $\bigwedge p\ m\ s.$
 $[\![$*inv-Move m*; *inv-Moves s*; *pre-play-move0 p m s*$]\!]$
 $\Longrightarrow$ *inv-Moves* $(s\ @\ [m])$
2. $\bigwedge p\ m\ s.$
 $[\![$*inv-Move m*; *inv-Moves s*; *pre-play-move0 p m s*$]\!] \Longrightarrow$ *pre-sum-elems s*
3. $\bigwedge p\ m\ s.$
 $[\![$*inv-Move m*; *inv-Moves s*; *pre-play-move0 p m s*$]\!]$
 $\Longrightarrow$ *pre-sum-elems* $(s\ @\ [m])$
4. $\bigwedge p\ m\ s.$
 $[\![$*inv-Move m*; *inv-Moves s*; *pre-play-move0 p m s*$]\!]$
 $\Longrightarrow$ *post-sum-elems s* (*sum-elems s*)
5. $\bigwedge p\ m\ s.$
 $[\![$*inv-Move m*; *inv-Moves s*; *pre-play-move0 p m s*$]\!]$
 $\Longrightarrow$ *post-sum-elems* $(s\ @\ [m])$ (*sum-elems* $(s\ @\ [m])$)
6. $\bigwedge p\ m\ s.$
 $[\![$*inv-Move m*; *inv-Moves s*; *pre-play-move0 p m s*$]\!]$
 $\Longrightarrow$ *sum-elems s* $<$ *sum-elems* $(s\ @\ [m])$
7. $\bigwedge p\ m\ s.$
 $[\![$*inv-Move m*; *inv-Moves s*; *pre-play-move0 p m s*$]\!]$
 $\Longrightarrow$ *sum-elems s* $+\ m =$ *sum-elems* $(s\ @\ [m])$

defer
apply (*simp add*: *l-sg2-pre-sume*) — SH, sg2

apply (*simp add*: *l-sg3-pre-sume-append*) — SH, sg3
apply (*simp add*: *l-inv-Moves-inv-SeqElems l-sg4-post-sume*) — SH, sg4
apply (*simp add*: *l-inv-Moves-inv-SeqElems l-sg5-post-sume-append*) — SH, sg5
apply (*simp add*: *l-inv-Move-nat1 l-sum-elems-append*) — SH, sg6
apply (*simp add*: *l-sum-elems-append*) — SH, sg7

1. $\bigwedge p\ m\ s.$
 $[\![$*inv-Move m*; *inv-Moves s*; *pre-play-move0 p m s*$]\!]$
 $\Longrightarrow$ *inv-Moves* $(s\ @\ [m])$

Yes! So, for the difficult case.

apply (*simp* (*no-asm*) *add*: *inv-Moves-def Let-def*, *intro conjI impI*)

1. $\bigwedge p\ m\ s.$
 $[\![$*inv-Move m*; *inv-Moves s*; *pre-play-move0 p m s*$]\!]$
 $\Longrightarrow$ *inv-SeqElems inv-Move* $(s\ @\ [m])$
2. $\bigwedge p\ m\ s.$
 $[\![$*inv-Move m*; *inv-Moves s*; *pre-play-move0 p m s*$]\!]$
 $\Longrightarrow$ *pre-sum-elems* $(s\ @\ [m])$
3. $\bigwedge p\ m\ s.$
 $[\![$*inv-Move m*; *inv-Moves s*; *pre-play-move0 p m s*$]\!]$
 $\Longrightarrow$ *post-sum-elems* $(s\ @\ [m])$ (*sum-elems* $(s\ @\ [m])$)
4. $\bigwedge p\ m\ s.$

$[\![$*inv-Move m*; *inv-Moves s*; *pre-play-move0 p m s*$]\!]$
$\Longrightarrow$ *sum-elems* $(s \mathbin{@} [m]) \leq$ *MAX-PILE*
5. $\bigwedge$*p m s*.
$[\![$*inv-Move m*; *inv-Moves s*; *pre-play-move0 p m s*;
sum-elems $(s \mathbin{@} [m]) =$ *MAX-PILE*$]\!]$
$\Longrightarrow$ *applyVDMSeq* $(s \mathbin{@} [m])$ $(len\ (s \mathbin{@} [m])) = 1$

As before, let us tackle each one of the sub parts in the definition

inv-Moves ?s $\equiv$
inv-SeqElems inv-Move ?s $\wedge$
pre-sum-elems ?s $\wedge$
(*let r* = *sum-elems ?s*
in post-sum-elems ?s r $\wedge$
$r \leq$ *MAX-PILE* $\wedge$ $(r =$ *MAX-PILE* $\longrightarrow$ *applyVDMSeq ?s* $(len\ ?s) = 1))$

oops

9.4.1 Handling (last?) difficult case on *inv-Moves* $(s \mathbin{@} [m])$

lemma *l-sg1-1-inv-Moves-seqelems-append*: *inv-Move m* $\Longrightarrow$ *inv-Moves ms* $\Longrightarrow$ *inv-SeqElems inv-Move* $(ms \mathbin{@} [m])$
by (*simp add*: *l-inv-Moves-inv-SeqElems l-inv-SeqElems-append*) — SH, subgoal 1.1

lemma *l-sg1-2-inv-Moves-pre-sume-append*: *inv-Move m* $\Longrightarrow$ *inv-Moves ms* $\Longrightarrow$ *pre-sum-elems* $(ms \mathbin{@} [m])$
by (*simp add*: *l-sg1-1-inv-Moves-seqelems-append l-pre-sume-seqelems-move*) — SH, subgoal 1.2

lemma *l-sg1-3-inv-Moves-post-sume-append*: *inv-Move m* $\Longrightarrow$ *inv-Moves ms* $\Longrightarrow$ *post-sum-elems* $(ms \mathbin{@} [m])$ (*sum-elems* $(ms \mathbin{@} [m]))$
by (*simp add*: *l-sg1-1-inv-Moves-seqelems-append l-sg5-post-sume-append*) — SH, subgoal 1.3

lemma *l-sg1-4-inv-Moves-maxpile-sume-append*: *inv-Move m* $\Longrightarrow$ *inv-Moves ms* $\Longrightarrow$ *sum-elems* $(ms \mathbin{@} [m]) \leq$ *MAX-PILE*
— nitpick quickcheck = none
apply (*simp add*: *l-sum-elems-append*)
apply (*induct ms*)
apply (*simp add*: *inv-Move-def*)
find-theorems *inv-Moves* (- # -)
apply (*frule l-inv-Moves-Hd*)
apply (*frule l-inv-Moves-Tl*)
apply *simp*

1. $\bigwedge$*a ms*.
$[\![m +$ *sum-elems ms* $\leq$ *MAX-PILE*; *inv-Move m*; *inv-Moves* $(a \mathbin{\#} ms)$;
inv-Move a; *inv-Moves ms*$]\!]$
$\Longrightarrow m + (a +$ *sum-elems ms*$) \leq$ *MAX-PILE*

We are stuck. Let us try the last subgoal.

oops

lemma *l-sg1-5-inv-Moves-last-move-append0*:
pre-play-move0 p m s $\Longrightarrow$ (*sum-elems* (*s* @ [*m*])) = *MAX-PILE* $\longrightarrow$ *applyVDMSeq* (*s* @ [*m*]) (*len* (*s* @ [*m*])) = *1*
using *l-applyVDMSeq-append-last l-sum-elems-append moves-left-def pre-play-move0-def*
by *force* — SH, subgoal 1.5

We are really narrowing it down: subgoal 1.4 has two subgoals, one we can finish. Let us set them up.

lemma *l-sg1-4-inv-Moves-moves-left-sume-append*: *pre-play-move0 p m ms* $\Longrightarrow$ *sum-elems* (*ms* @ [*m*]) $\leq$ *MAX-PILE*
unfolding *pre-play-move0-def*
apply (*elim conjE impE*)

1. ⟦*inv-Move m*; *inv-Moves ms*; *pre-moves-left ms*; *pre-fair-play p ms*;
   post-fair-play p ms (*fair-play p ms*); *0* < *moves-left ms*;
   fair-play p ms⟧
   $\Longrightarrow$ *moves-left ms* $\neq$ *1*
2. ⟦*inv-Move m*; *inv-Moves ms*; *pre-moves-left ms*; *pre-fair-play p ms*;
   post-fair-play p ms (*fair-play p ms*); *0* < *moves-left ms*;
   fair-play p ms; *m* < *moves-left ms*⟧
   $\Longrightarrow$ *sum-elems* (*ms* @ [*m*]) $\leq$ *MAX-PILE*

defer
apply (*simp add*: *l-sum-elems-append moves-left-def*) — SH, subgoal 1.4.2
oops

lemma *l-sg1-4-2-inv-Moves-moves-left-sume-append*:
inv-Move m $\Longrightarrow$ *inv-Moves ms* $\Longrightarrow$ *m* < *moves-left ms* $\Longrightarrow$ *sum-elems* (*ms* @ [*m*]) $\leq$ *MAX-PILE*
by (*simp add*: *l-sum-elems-append moves-left-def*) — SH, subgoal 1.4.2

9.4.2 Getting to the missing terms in *pre-play-move*

lemma *l-sg1-4-1-inv-Moves-moves-left-sume-append*:
inv-Move m $\Longrightarrow$ *inv-Moves ms* $\Longrightarrow$ *0* < *moves-left ms* $\Longrightarrow$ *moves-left ms* $\neq$ *1*
— nitpick quickcheck = none
unfolding *moves-left-def inv-Move-def inv-VDMNat1-def* **apply** (*elim conjE*, *intro notI*)
oops

To try and understand what is the problem, we generalise the expressions to simpler terms. And get to the following unprovable conjecture, and its improved version.

lemma *l-sg1-4-1-explore*: $x \leq$ *MAX-MOV* $\Longrightarrow$ $y \leq$ *MAX-PILE* $\Longrightarrow$ $x + y \leq$ *MAX-PILE*
nitpick[*user-axioms*]

oops

lemma *l-sg1-4-1-inv-Moves-maxpile-moves-left-gen*:
$x \leq \textit{MAX-MOV} \Longrightarrow y \leq \textit{MAX-PILE} \Longrightarrow x < \textit{MAX-PILE} - y \Longrightarrow x + y \leq \textit{MAX-PILE}$
by *auto*

9.4.3 Proving the *inv-Moves* $(s \mathbin{@} [m])$ subgoal

With the new *pre-play-move*, we can now collect all lemmas again for the top-level proof. On all subgoals, only 1.4.1 needed the new definition. Yet, *pre-play-move0* fetured in subgoal 1.5. We need to redefine it with the new precondition. Also, if using the *pre-play-move* as an assumption it will not match with the goal after expansion.

lemma *l-sg1-5-inv-Moves-last-move-append*:
pre-play-move p m s $\Longrightarrow$ $(\textit{sum-elems}\ (s \mathbin{@} [m])) = \textit{MAX-PILE} \longrightarrow \textit{applyVDMSeq}\ (s \mathbin{@} [m])\ (\textit{len}\ (s \mathbin{@} [m])) = 1$
using *l-applyVDMSeq-append-last l-sum-elems-append moves-left-def pre-play-move-def*
by *force* — SH, subgoal 1.5

Given the change to *pre-play-move*, we also add a lemma that a previously state postcondition is now a direct consequence of the current post condition.

lemma *l-pre-play-moves-left-nat1*:
pre-play-move p m s $\Longrightarrow$ *moves-left s* > 0
using *pre-play-move-def l-inv-Move-nat1* **by** *fastforce* — SH

lemma *l-sg1-4-inv-Moves-moves-left-sume-append*:
pre-play-move p m ms $\Longrightarrow$ $\textit{sum-elems}\ (ms \mathbin{@} [m]) \leq \textit{MAX-PILE}$
unfolding *pre-play-move-def*
apply (*simp only*: *le-less*)
apply (*simp* (*no-asm*) *only*: *le-less*[*symmetric*])
apply (*elim conjE disjE*)

1. $[\![$*inv-Move m*; *inv-Moves ms*; *pre-moves-left ms*; *pre-fair-play p ms*;
   post-fair-play p ms (*fair-play p ms*); *fair-play p ms*;
   $\textit{moves-left ms} = m \longrightarrow m = 1$; $m < \textit{moves-left ms}]\!]$
   $\Longrightarrow \textit{sum-elems}\ (ms \mathbin{@} [m]) \leq \textit{MAX-PILE}$
2. $[\![$*inv-Move m*; *inv-Moves ms*; *pre-moves-left ms*; *pre-fair-play p ms*;
   post-fair-play p ms (*fair-play p ms*); *fair-play p ms*;
   $\textit{moves-left ms} = m \longrightarrow m = 1$; $m = \textit{moves-left ms}]\!]$
   $\Longrightarrow \textit{sum-elems}\ (ms \mathbin{@} [m]) \leq \textit{MAX-PILE}$

apply (*simp add*: *l-sg1-4-2-inv-Moves-moves-left-sume-append*) — SH, sg 1.4.2
by (*simp add*: *l-sum-elems-append moves-left-def*) — SH, sg 1.4.1

Now we can prove the first subgoal

lemma *l-sg1-inv-Moves-append*:
pre-play-move p m s $\Longrightarrow$ *inv-Moves* $(s \mathbin{@} [m])$

unfolding *inv-Moves-def pre-play-move-def Let-def*
apply (*elim conjE*, *intro conjI impI*, *simp-all*)
apply (*simp add*: *l-sg1-1-inv-Moves-seqelems-append pre-moves-left-def*) — SH, 1.1
apply (*simp add*: *l-sg1-2-inv-Moves-pre-sume-append pre-moves-left-def*) — SH, 1.2
apply (*simp add*: *l-sg1-3-inv-Moves-post-sume-append pre-moves-left-def*)— SH, 1.3
apply (*metis* (*full-types*) *l-sg1-4-inv-Moves-moves-left-sume-append pre-fair-play-def pre-play-move-def*) — SH, 1.4
by (*smt l-sg1-5-inv-Moves-last-move-append pre-fair-play-def pre-play-move-def*) — SH, 1.5

9.5 Putting it all together

3 Lemma-based attempt with sledgehammer support.

definition
PO-play-move-sat-obl :: $\mathbb{B}$
where
PO-play-move-sat-obl $\equiv \forall$ *p m s* . *inv-Move m* $\longrightarrow$ *inv-Moves s* $\longrightarrow$
pre-play-move p m s $\longrightarrow$ ($\exists$ *r* . *post-play-move p m s r*)

definition
PO-play-move-sat-exp-obl :: $\mathbb{B}$
where
PO-play-move-sat-exp-obl $\equiv \forall$ *p m s* . *inv-Move m* $\longrightarrow$ *inv-Moves s* $\longrightarrow$
pre-play-move p m s $\longrightarrow$ *post-play-move p m s* (*play-move p m s*)

And finally, we have all the lemmas we need to prove the satisfiability of *play-move*.

theorem *PO-play-move-sat-exp-obl*

...

1. $\bigwedge$*p m s*.
   $[\![$*inv-Move m*; *inv-Moves s*; *pre-play-move p m s*$]\!] \Longrightarrow$ *inv-Moves* (*s* @ [*m*])
2. $\bigwedge$*p m s*.
   $[\![$*inv-Move m*; *inv-Moves s*; *pre-play-move p m s*$]\!] \Longrightarrow$ *pre-sum-elems s*
3. $\bigwedge$*p m s*.
   $[\![$*inv-Move m*; *inv-Moves s*; *pre-play-move p m s*$]\!]$
   $\Longrightarrow$ *pre-sum-elems* (*s* @ [*m*])
4. $\bigwedge$*p m s*.
   $[\![$*inv-Move m*; *inv-Moves s*; *pre-play-move p m s*$]\!]$
   $\Longrightarrow$ *post-sum-elems s* (*sum-elems s*)
5. $\bigwedge$*p m s*.
   $[\![$*inv-Move m*; *inv-Moves s*; *pre-play-move p m s*$]\!]$
   $\Longrightarrow$ *post-sum-elems* (*s* @ [*m*]) (*sum-elems* (*s* @ [*m*]))
6. $\bigwedge$*p m s*.
   $[\![$*inv-Move m*; *inv-Moves s*; *pre-play-move p m s*$]\!]$
   $\Longrightarrow$ *sum-elems s* $<$ *sum-elems* (*s* @ [*m*])
7. $\bigwedge$*p m s*.
   $[\![$*inv-Move m*; *inv-Moves s*; *pre-play-move p m s*$]\!]$
   $\Longrightarrow$ *sum-elems s* $+$ *m* $=$ *sum-elems* (*s* @ [*m*])

apply (*simp add*: *l-sg1-inv-Moves-append*) — SH, sg1
apply (*simp add*: *l-sg2-pre-sume*) — SH, sg2
apply (*simp add*: *l-sg3-pre-sume-append*) — SH, sg3
apply (*simp add*: *l-inv-Moves-inv-SeqElems l-sg4-post-sume*) — SH, sg4
apply (*simp add*: *l-inv-Moves-inv-SeqElems l-sg5-post-sume-append*) — SH, sg5
apply (*simp add*: *l-inv-Move-nat1 l-sum-elems-append*) — SH, sg6
by (*simp add*: *l-sum-elems-append*) — SH, sg7

10 VDM Operations satisfiability POs

theorem *PO-first-player-winning-choose-move-sat-exp-obl*
unfolding *PO-first-player-winning-choose-move-sat-exp-obl-def*
apply (*intro allI impI*)
unfolding *pre-first-player-winning-choose-move-def*
post-first-player-winning-choose-move-def
apply (*elim conjE*)
unfolding *post-fixed-choose-move-def*
apply *simp*
apply (*intro conjI*)
unfolding *inv-Move-def max-def Let-def*
apply *simp*

oops

Intermediate result needed for first subgoal. Also create the structured expansion as *lemmas* statements.

lemma *l-best-move-range*: *best-move ms* $\geq$ *1* $\Longrightarrow$ *best-move ms* $\leq$ *MAX-MOV*
unfolding *best-move-def moves-left-def* **by** *simp*

lemma *l-best-move-nat*: *0* $\leq$ *best-move ms*
unfolding *best-move-def* **by** *simp*

lemma *l-best-move-nat1*: *inv-Moves ms* $\Longrightarrow$ (*0* $<$ *best-move ms*) $=$ *will-first-player-win*
oops

lemmas *PO-first-player-winning-choose-move-sat-exp-obl-pre-post* =
PO-first-player-winning-choose-move-sat-exp-obl-def
pre-first-player-winning-choose-move-def
post-first-player-winning-choose-move-def
post-fixed-choose-move-def

lemma *l-first-player-win-best-move*: *inv-Move* (*max 1* (*best-move ms*))
using *inv-Move-def inv-Move-defs*(2) *l-best-move-range* **by** *force* — SH

theorem *PO-first-player-winning-choose-move-sat-exp-obl*
unfolding *PO-first-player-winning-choose-move-sat-exp-obl-pre-post*
apply (*intro allI impI*, *elim conjE*, *intro conjI*, *simp-all*)
apply (*simp add*: *l-first-player-win-best-move*)

unfolding *inv-Move-def inv-VDMNat1-def max-def Let-def*
apply (*simp add*: *l-best-move-range*)
apply (*intro conjI impI*)

oops

Deduce information from *inv-Nim* without the need to expand it

lemmas *inv-Nim-defs* = *inv-Nim-def inv-Nim-flat-def*
lemma *f-Nim-inv-Moves*: *inv-Nim st* $\Longrightarrow$ *inv-Moves* (*moves st*)
unfolding *inv-Nim-defs* **by** *simp*

lemma *l-isFirst*: *isFirst P1*
unfolding *isFirst-def* **by** *simp*

thm *Let-def option.split split-if*

lemma *l-moves-left-sat*: *pre-moves-left ms* $\Longrightarrow$ *post-moves-left ms* (*moves-left ms*)
by (*meson inv-Moves-def l-inv-VDMNat-moves-left post-moves-left-def pre-moves-left-def*)
— SH

lemma *l-play-move-sat*: *pre-play-move0 p m ms* $\Longrightarrow$ *post-play-move p m ms* (*play-move p m ms*)
unfolding *pre-play-move0-def post-play-move-def*
apply (*elim conjE*, *simp*, *intro conjI*)
oops

lemma *l-play-move-inv-moves*: *inv-Move m* $\Longrightarrow$ *inv-Moves ms* $\Longrightarrow$ *pre-play-move0 p m ms* $\Longrightarrow$ *inv-Moves* (*play-move p m ms*)
unfolding *inv-Moves-defs play-move-def pre-play-move0-def Let-def*
apply (*simp add*: *l-applyVDMSeq-append-last*)
apply (*simp add*: *l-sum-elems-append l-len-append*)
apply (*elim conjE*, *intro conjI impI*)
using *inv-VDMNat-def l-inv-Move-nat1* **apply** *force* — SH
using *l-inv-Move-nat1* **apply** *force* — SH
unfolding *Let-def*
oops

theorem *PO-first-player-winning-choose-move-sat-exp-obl*
unfolding *PO-first-player-winning-choose-move-sat-exp-obl-pre-post*
apply (*intro allI impI*, *elim conjE*, *intro conjI*, *simp-all*)
unfolding *inv-Move-def max-def*
apply (*simp add*: *l-best-move-range*)
unfolding *pre-who-plays-next-def Let-def*
apply (*simp add*: *inv-VDMNat1-def*)
unfolding *pre-play-move-def*
apply (*simp*)

oops

Generalise *l-best-move-range* to avoid expanding *inv-Move*. Notice that the condition for the theorem needs to be as it appears in the goals.

lemma *l-best-move-range2*: $1 \leq$ *best-move* (*moves st*) $\Longrightarrow$ *inv-Move* (*best-move* (*moves st*))
unfolding *inv-Move-defs best-move-def moves-left-def* **by** (*simp*)

theorem *PO-first-player-winning-choose-move-sat-exp-obl*
unfolding *PO-first-player-winning-choose-move-sat-exp-obl-pre-post*
apply (*intro allI impI*, *elim conjE*, *intro conjI*, *simp-all*)
unfolding *max-def*
apply (*simp add*: *l-best-move-range2*)

oops

lemma *PO-first-player-winning-choose-move-sat-exp-obl*
unfolding *PO-first-player-winning-choose-move-sat-exp-obl-pre-post*
apply (*intro allI impI*, *elim conjE*, *intro conjI*, *simp-all*)
unfolding *max-def*
apply (*smt l-first-player-win-best-move*)
unfolding *Let-def*
apply (*simp add*: *l-best-move-range2 l-inv-Move-nat1*)
unfolding *pre-who-plays-next-def*
apply (*simp add*: *f-Nim-inv-Moves l-isFirst*)
unfolding *pre-play-move-def*
apply *simp*
 apply (*intro impI conjI*)
 apply (*simp-all add*: *l-best-move-range2 l-inv-Move-nat1 f-Nim-inv-Moves*)

oops

Property about *best-move* and *moves-left*. Is it true? Are there conditions?

lemma *l-best-move-inv*: *inv-Nim st* $\Longrightarrow$ *best-move s* $<$ *moves-left s*
find-theorems *name*:*sum-elems*
unfolding *best-move-def moves-left-def*
apply *simp*
find-theorems *name*:*induct name*:*Nat*
apply (*induct sum-elems s*)

oops

lemma *PO-first-player-winning-choose-move-sat-obl*
unfolding *PO-first-player-winning-choose-move-sat-obl-def pre-first-player-winning-choose-move-def post-first-player-winning-choose-move-def*
apply (*intro allI impI*, *elim conjE*)
unfolding *max-def*
apply (*simp add*: *l-best-move-range2*)
unfolding *pre-who-plays-next-def*
apply (*simp add*: *l-inv-Move-nat1 l-isFirst*)
unfolding *pre-moves-left-def*
apply (*simp add*: *l-isFirst*)

oops

Let us try the lemma about *best-move* again, but generalise it this time. Say, take the expression:

$$\textit{best-move ms} < \textit{moves-left ms}[display = true] = (\textit{moves-left ms} - \mathit{1})\ \textit{mod}\ (\textit{MAX-MOV} + \mathit{1}) < \textit{moves-left}$$

Now, let us investigate known facts about *x mod y* under $\mathbb{N}$.

quickcheck immediately finds the useful counter examples, which if ruled out by suitable assumptions on involved values leads to the main result discovered by sledgehammer.

lemma *l-best-move-mov-limit-mod*: $n > 0 \Longrightarrow m > 0 \Longrightarrow ((m{::}int) - 1)\ mod\ n < m$

using *zle-diff1-eq zmod-le-nonneg-dividend* **by** *blast*

lemma *l-best-move-inv*: *moves-left s* $> 0 \Longrightarrow$ *best-move s* $<$ *moves-left s*
unfolding *best-move-def*
using [[*rule-trace*,*simp-trace*]]
by (*simp only*: *l-best-move-mov-limit-mod*)

To be continued...

end