CSC3323 Isabelle Tutorials

By Leo Freitas

March 18, 2021

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-MAX-PILE $>$ *0*
and *G-MAX-MOV* $>$ *0*
and *G-MAX-MOV* $<$ *G-MAX-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* | *P2*

definition
inv-Player :: *Player* $\Rightarrow \mathbb{B}$
where
inv-Player p $\equiv$ *True*

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-Move :: *VDMNat1* ⇒ $\mathbb{B}$
where
inv-Move m ≡ *inv-VDMNat1 m* ∧ *m* ≤ *MAX-MOV*

value *inv-Move 3*
value *inv-Move 5*

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 uses a quantifier instead of *inv-SeqElems*.

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.

type-synonym *Moves0* = *Move VDMSeq*

value ([*10*,*11*,*12*,*20*]::*Moves0*) ! *1*

value ([*10*,*11*,*12*,*20*]::*Moves0*) $ *1*

value [*1*,*1*,*2*,*10*]::*Moves0*

definition
inv-Moves0 :: *Moves0* ⇒ $\mathbb{B}$
where
inv-Moves0 m ≡ ∀ *i* ∈ *inds m* . *inv-Move* (*m* $ *i*)

value *inv-Moves0* [*1*,*1*,*2*,*1*,*2*,*3*]

definition
inv-Moves0-new :: *Moves0* ⇒ $\mathbb{B}$
where
inv-Moves0-new m ≡ *inv-SeqElems inv-Move m*

lemma *inv-Moves0-new s* = *inv-Moves0 s*
unfolding *inv-Moves0-new-def inv-Moves0-def*
apply (*induct s*, *simp-all*)

apply (*simp add*: *inv-SeqElems-def*)
oops

find-theorems *inv-SeqElems* - -

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*

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.

declare [[*show-types*]]
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*

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

 apply (*frule l-sum-elems-nat*)
 apply *simp*

 1. ⋀(*a*::ℤ) *s*::ℤ *list*.
   ⟦⟦*inv-SeqElems inv-Move s*; *s* ≠ []⟧ ⟹ (*0*::ℤ) < *sum-elems s*;
   inv-SeqElems inv-Move (*a* # *s*); (*0*::ℤ) ≤ *a* + *sum-elems s*⟧
   ⟹ (*0*::ℤ) < *a* + *sum-elems s*
variables:
 s :: ℤ *list*

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*
unfolding *inv-SeqElems-def*
apply (*induct s*)

apply *simp*
proof –
  fix $a :: \mathbb{Z}$ **and** $s :: \mathbb{Z}$ *list*
  assume *a1*: *list-all inv-Move* $(a \# s)$
  assume *a2*: $[\![$*list-all inv-Move s*; $s \neq []]\!] \Longrightarrow 0 <$ *sum-elems s*
  have *f3*: $\bigwedge$*x1 e-x*. *sum-elems x1* + *e-x* = *sum-elems* (*e-x* # *x1*) **by** (*simp add*: *add.commute*)
  have $\bigwedge$*x1*. *sum-elems* [*x1*] = *x1* **by** *simp*
  thus 0 < *sum-elems* (*a* # *s*) **using** *a1 a2 f3* **by** (*metis* (*no-types*) *add-mono-thms-linordered-field*(5) *l-inv-Move-nat1 list.pred-inject*(2) *monoid-add-class.add.right-neutral*)
qed

lemma *l-sum-elems-notempty*:
  inv-SeqElems inv-Move s $\Longrightarrow$ 0 < *sum-elems s* $\Longrightarrow s \neq []$ **by** *auto*

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

   inv-SeqElems inv-Move (*?s*::$\mathbb{Z}$ *list*) $\Longrightarrow$ (0::$\mathbb{Z}$) $\leq$ *sum-elems ?s*

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

   $[\![$*inv-SeqElems inv-Move* (*?s*::$\mathbb{Z}$ *list*); *?s* $\neq []]\!] \Longrightarrow$ (0::$\mathbb{Z}$) < *sum-elems ?s*

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

   $[\![$*inv-SeqElems inv-Move* (*?s*::$\mathbb{Z}$ *list*); (0::$\mathbb{Z}$) < *sum-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 $\equiv$
    pre-sum-elems s $\longrightarrow$
    (*inv-VDMNat RESULT* $\wedge$
     ($s \neq [] \longleftrightarrow$ *RESULT* > 0))

lemma $\forall$ *s* . *pre-sum-elems s* $\longrightarrow$ ($\exists$ *r* . *post-sum-elems s r*)
by (*metis inv-VDMNat-def le-cases n1-MP not-le post-sum-elems-def*)

lemma $\forall\ s\ .\ \textit{pre-sum-elems}\ s \longrightarrow (\textit{post-sum-elems}\ s\ (\textit{sum-elems}\ s))$
by (*metis NimFull.sum-elems0 inv-VDMNat-def l-sum-elems-nat1 le-less less-irrefl post-sum-elems-def pre-sum-elems-def*)

Useful properties about *sum-elems* specification.

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

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

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 *sum-elems*. For instance, the theorem for the explicitly defined *sum-elems* function is:

$\forall\ s{::}\mathbb{Z}\ \textit{list}.$
$\quad \textit{inv-SeqElems inv-Move}\ s \longrightarrow$
$\quad \textit{pre-sum-elems}\ s \longrightarrow \textit{post-sum-elems}\ s\ (\textit{sum-elems}\ s)$

That is, given any valid input value (*inv-SeqElems inv-Move* ($ms{::}\mathbb{Z}$ *list*)), 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.

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.* $\textit{expr}{::}'a$) must feature in the proof. This will be particularly interesting in the proof below about $\textit{best-move}{::}'a$, where the general case is provable, whereas the one with the initial explicit definition of $\textit{best-move}{::}'a$ is not. **That is, the specification is feasible for some implementation but not the one given by the explicit definition!**

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

definition
$\textit{PO-sum-elems-sat-exp-obl} :: \mathbb{B}$
where
$\textit{PO-sum-elems-sat-exp-obl} \equiv \forall\ s\ .\ \textit{inv-SeqElems inv-Move}\ s \longrightarrow$

pre-sum-elems s $\longrightarrow$ *post-sum-elems s* (*sum-elems s*)

lemma *PO-sum-elems-sat-exp-obl*
by (*metis NimFull.sum-elems0 PO-sum-elems-sat-exp-obl-def inv-VDMNat-def l-pre-sum-elems-sat le-less post-sum-elems-def*)

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 *safe*
 apply (*rule-tac x=sum-elems s* **in** *exI*)
 apply *safe*
  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)

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)

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*::*'a* 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*::*'a list*) ! (*i*::$\mathbb{N}$). 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 (*s*::*'a list*) \$ (*x*::$\mathbb{Z}$)), 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 $[a,b]$ \$ $(len\ [a,b])$

type-synonym $Moves = Move\ VDMSeq$

definition
$inv\text{-}Moves :: Moves \Rightarrow \mathbb{B}$
where
$inv\text{-}Moves\ s \equiv$
$\quad inv\text{-}SeqElems\ inv\text{-}Move\ s \wedge$
$\quad pre\text{-}sum\text{-}elems\ s \wedge$
$\quad (let\ r = sum\text{-}elems\ s\ in$
$\quad\ post\text{-}sum\text{-}elems\ s\ r \wedge$
$\quad\ r \leq MAX\text{-}PILE \wedge$
$\quad\ (r = MAX\text{-}PILE \longrightarrow s\ \$\ (len\ s) = 1))$

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*::'*a*

```
-- 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
$who\text{-}plays\text{-}next\ ms \equiv (if\ (len\ ms)\ vdmmod\ 2 = 0\ then\ P1\ else\ P2)$

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

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
$pre\text{-}who\text{-}plays\text{-}next :: Moves \Rightarrow \mathbb{B}$
where
$pre\text{-}who\text{-}plays\text{-}next\ ms \equiv inv\text{-}Moves\ ms \wedge isFirst\ P1$

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

5.1.2 Satisfiability PO

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

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

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

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

5.2 *fair-play*::*'a*

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

Notice that in Isabelle, we get curried definitions (e.g. $\mathit{fair\text{-}play}{::}'a$ is called as $(\mathit{fair\text{-}play}{::}'a \Rightarrow {}'b \Rightarrow {}'c)\ (p{::}'a)\ (ms{::}'b))$ for VDM functions with multiple parameters.

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

5.2.1 Specification

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

definition
$\mathit{post\text{-}fair\text{-}play} :: \mathit{Player} \Rightarrow \mathit{Moves} \Rightarrow \mathbb{B} \Rightarrow \mathbb{B}$
where
$\mathit{post\text{-}fair\text{-}play}\ p\ ms\ \mathit{RESULT} \equiv$
$\quad \mathit{pre\text{-}fair\text{-}play}\ p\ ms \longrightarrow \mathit{post\text{-}who\text{-}plays\text{-}next}\ ms\ p$

5.2.2 Satisfiability PO

definition
$\mathit{PO\text{-}fair\text{-}play\text{-}sat\text{-}obl} :: \mathbb{B}$
where
$\mathit{PO\text{-}fair\text{-}play\text{-}sat\text{-}obl} \equiv \forall\ s\ p\ .\ \mathit{inv\text{-}Moves}\ s \longrightarrow$
$\quad \mathit{pre\text{-}fair\text{-}play}\ p\ s \longrightarrow (\exists\ r\ .\ \mathit{post\text{-}fair\text{-}play}\ p\ s\ r)$

theorem *PO-fair-play-sat-obl*
by (*simp add*: *PO-fair-play-sat-obl-def inv-Player-def post-fair-play-def post-who-plays-next-def*)

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

theorem *PO-fair-play-sat-exp-obl*
by (*simp add*: *PO-fair-play-sat-exp-obl-def inv-Player-def post-fair-play-def post-who-plays-next-def*)

5.3 $\mathit{moves\text{-}left}{::}'a$

```
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*

definition
post-moves-left :: *Moves* ⇒ *VDMNat* ⇒ $\mathbb{B}$
where
post-moves-left ms RESULT ≡
pre-moves-left ms ⟶
inv-VDMNat RESULT ∧
post-sum-elems ms (*sum-elems ms*)

5.3.2 Satisfiability PO

definition
PO-moves-left-sat-obl :: $\mathbb{B}$
where
PO-moves-left-sat-obl ≡ ∀ *s* . *inv-Moves s* ⟶
pre-moves-left s ⟶ (∃ *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)

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

theorem *PO-moves-left-sat-exp-obl*
unfolding *PO-moves-left-sat-exp-obl-def inv-Moves-def inv-VDMNat-def*
moves-left-def post-moves-left-def Let-def
by *simp*

5.4 *vdmtake*

The function *take* has been defined for lists in Isabelle. We use the VDMToolkit vdmtake instead.

value *vdmtake 2* [*A*,*B*]
value *take 2* [*A*,*B*]

5.5 *seq-prefix*

The function *seq-prefix* has been defined inside VDMToolkit.

value $[(1{::}nat),2] \sqsubseteq [1,2,3]$
value $[(1{::}nat),2] \sqsubseteq [1,3,2]$

5.6 *play-move*::'*a*

```
play_move: Player * Move * Moves -> Moves
play_move(p, m, s) == s ^ [m]
pre
  -- x cannot play to loose, but at the end
  -- x cannot play to loose before the end
  (moves_left(s) <> 1 => m < moves_left(s))
  and
  -- must play to loose at the end
  (moves_left(s) = m => m = 1)
  and
  --there must be something to be played
  --moves_left(s) > m
  --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)
  and
  not fair_play(p, RESULT)
  and
  seq_prefix[Move](s, RESULT)
```

definition
play-move :: *Player* ⇒ *Move* ⇒ *Moves* ⇒ *Moves*
where
play-move p m s ≡ *s* @ [*m*]

5.6.1 Specification

definition
pre-play-move0 :: *Player* ⇒ *Move* ⇒ *Moves* ⇒ $\mathbb{B}$
where
pre-play-move0 p m s ≡
$\quad$ *inv-Move m* ∧ *inv-Moves s* ∧ *pre-moves-left s* ∧ *pre-fair-play p s* ∧

$post\text{-}fair\text{-}play\ p\ s\ (fair\text{-}play\ p\ s) \wedge$
$(moves\text{-}left\ s \neq 1 \longrightarrow m < moves\text{-}left\ s) \wedge$
$0 < moves\text{-}left\ s \wedge fair\text{-}play\ p\ s$

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

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

5.6.2 Satisfiability PO

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

5.7 *will-first-player-win*::'*a*

```
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 $(x::\mathbb{Z} \Rightarrow 'a)\ (-\ (1::\mathbb{Z}))$ and $(x::\mathbb{Z}) - (1::\mathbb{Z})$: 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
$will\text{-}first\text{-}player\text{-}win :: Move \Rightarrow \mathbb{B}$
where
$will\text{-}first\text{-}player\text{-}win\ limit \equiv (MAX\text{-}PILE - 1)\ vdmmod\ (limit + 1) \neq 0$

5.7.1 Specification

definition
pre-will-first-player-win :: *Move* $\Rightarrow \mathbb{B}$
where
pre-will-first-player-win limit $\equiv$ *inv-MAX-PILE* $\wedge$ *inv-Move limit* $\wedge$ *pre-vdm-mod* (*MAX-PILE* $-$ *1*) (*limit* $+$ *1*)

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

5.7.2 Satisfiability PO

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

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

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

theorem *PO-will-first-player-win-sat-exp-obl*
unfolding *PO-will-first-player-win-sat-exp-obl-def*
will-first-player-win-def pre-will-first-player-win-def will-first-player-win-def
pre-vdm-mod-def inv-Move-def inv-VDMNat1-def inv-MAX-PILE-def
apply *simp*
apply (*intro allI impI*, *elim conjE*)

proof of specific/individual cases is more difficult, paradoxically

apply (*case-tac limit*, *simp-all*)
apply (*case-tac n=1*, *simp-all*)
apply (*case-tac n=2*, *simp-all*)
by (*case-tac n=3*, *simp-all*)

5.8 *who-won-invariant*::*'a*

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

5.8.1 Specification

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

definition
post-who-won-invariant :: $Player \Rightarrow Moves \Rightarrow Move \Rightarrow \mathbb{B} \Rightarrow \mathbb{B}$
where
post-who-won-invariant winner moves limit RESULT $\equiv$
pre-who-won-invariant winner moves limit $\longrightarrow$
post-moves-left moves (*moves-left moves*) $\wedge$
post-who-plays-next moves winner

5.8.2 Satisfiability PO

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

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

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*
apply *simp*
oops

5.9 *first-player*::$'a$

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

definition
first-player :: *Player*
where
first-player $\equiv$ (*if isFirst P1 then P1 else P2*)

5.9.1 Specification

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

5.9.2 Satisfiability PO

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

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

5.10 *first-player-inds*::$'a$

```
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* ⇒ *VDMNat1 VDMSet*
where
first-player-inds ms ≡ { *i* | *i* . *i* ∈ *inds 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*::$'a$ command.

value {*i* . *i*∈ {(*0*::*int*),*1*,*2*,*3*}}
value {(*i*,*i*)| *i* . *i*∈ {(*0*::*int*),*1*,*2*,*3*} }

lemma *A* = {*i* . *i*∈ {(*0*::*int*),*1*,*2*,*3*} | *i* < *2*} **apply** *simp* **oops**
lemma *A* = {*i* | *i* . *i*∈ {(*0*::*int*),*1*,*2*,*3*} ∧ *i* < *2*} **apply** *simp* **oops**
lemma {*0*,*1*} = {*i* | *i* . *i*∈ {(*0*::*int*),*1*,*2*,*3*} ∧ *i* < *2*} **apply** *auto* **done**

5.10.1 Specification

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

definition
post-first-player-inds :: *Moves* ⇒ *VDMNat1 VDMSet* ⇒ 𝔹
where
post-first-player-inds ms RESULT ≡ *inv-Moves ms* ∧
inv-SetElems inv-VDMNat1 RESULT ∧ *RESULT* ⊆ *inds ms*

5.10.2 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*

definition
PO-first-player-inds-sat-exp-obl :: $\mathbb{B}$
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 inv-VDMNat1* (*first-player-inds s*)
unfolding *first-player-inds-def inds-as-nat-def len-def inv-SetElems-def inv-VDMNat1-def*
unfolding *inds-def*
by (*simp*)

lemma *l-first-player-inds-within-inds*:
first-player-inds s ⊆ *inds s*
unfolding *first-player-inds-def inds-as-nat-def len-def inv-SetElems-def inv-VDMNat1-def*
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*)

5.11 *moves-of* :: $'a$

```
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 $[\ [a,b,c]\ !\ i\ .\ i \leftarrow [2,1,0]]$

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

definition
moves-of :: *Moves* ⇒ 𝔹 ⇒ *Move VDMSeq*
where
moves-of ms first ≡
  (*let idxs* = *first-player-inds ms in*
   [*ms* ! (*nat i*) . *i* ← *sorted-list-of-set* (*if first then idxs else* (*inds ms* − *idxs*))])

5.11.1 Specification

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

definition
post-moves-of :: *Moves* ⇒ 𝔹 ⇒ *Move VDMSeq* ⇒ 𝔹
where
post-moves-of ms first RESULT ≡
  inv-Moves ms ∧ *inv-SeqElems inv-Move RESULT* ∧
  pre-first-player-inds ms ∧ *post-first-player-inds ms* (*first-player-inds ms*)

5.11.2 Satisfiability PO

definition
PO-moves-of-sat-obl :: 𝔹
where
PO-moves-of-sat-obl ≡
  ∀ *s f* . *pre-moves-of s f* ⟶ (∃ *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 :: 𝔹
where
PO-moves-of-sat-exp-obl ≡ ∀ *s f* . *inv-Moves s* ⟶
  pre-moves-of s f ⟶ *post-moves-of s f* (*moves-of s f*)

lemma *l-moves-of-move*:
inv-Moves ms ⟹ *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*)
apply (*simp add*: *l-first-player-inds-within-inds*)
oops

5.12 *best-move*::$'a$

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

definition
best-move :: *Moves* $\Rightarrow$ *VDMNat*
where
best-move moves $\equiv$ ((*moves-left moves*) $-$ *1*) *vdmmod* (*MAX-MOV* $+$ *1*)

5.12.1 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 (0::$\mathbb{Z}$) $<$ *moves-left* (*ms*::$\mathbb{Z}$ *list*), which prevents modulo arithmetic over negative numbers, whereas the first post condition used the wrong specification of *post-moves-left0*::$'a$.

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

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

RESULT $\leq$ *moves-left ms*

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

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

5.12.2 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:

$\forall$ *ms*::$\mathbb{Z}$ *list*.
inv-Moves ms $\longrightarrow$ *pre-best-move ms* $\longrightarrow$ ($\exists$ *r*::$\mathbb{Z}$. *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*)

value *3* +(*5*::*int*)

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*

theorem *PO-best-move-sat-obl0*
unfolding *PO-best-move-sat-obl0-def*
pre-best-move0-def post-best-move0-def

find-theorems $\forall$ - . (- $\longrightarrow$ -)
apply *simp*
unfolding *pre-moves-left-def post-moves-left-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. $\bigwedge ms{::}\mathbb{Z}\ list.$
   $[\![inv\text{-}Moves\ ms; inv\text{-}SeqElems\ inv\text{-}Move\ ms]\!] \Longrightarrow (0{::}\mathbb{Z}) \leq moves\text{-}left\ ms$
2. $\bigwedge ms{::}\mathbb{Z}\ list.$
   $[\![inv\text{-}Moves\ ms; inv\text{-}SeqElems\ inv\text{-}Move\ ms]\!] \Longrightarrow (0{::}\mathbb{Z}) \leq sum\text{-}elems\ ms$
3. $\bigwedge ms{::}\mathbb{Z}\ list.$
   $[\![inv\text{-}Moves\ ms; inv\text{-}SeqElems\ inv\text{-}Move\ ms]\!]$
   $\Longrightarrow (ms \neq []) = ((0{::}\mathbb{Z}) < sum\text{-}elems\ ms)$
4. $\bigwedge ms{::}\mathbb{Z}\ list.$
   $[\![inv\text{-}Moves\ ms; inv\text{-}SeqElems\ inv\text{-}Move\ ms]\!] \Longrightarrow (0{::}\mathbb{Z}) \leq moves\text{-}left\ ms$

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

prefer *2*
 apply (*simp add*: *l-sum-elems-nat*)
prefer *2*
using *l-sum-elems-nat1* **apply** *auto*[*1*]
 apply (*simp add*: *l-moves-left-nat*)+
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-nat1*:
 $inv\text{-}Moves\ ms \Longrightarrow 0 < moves\text{-}left\ ms$
apply (*induct ms*)
unfolding *moves-left-def*
apply *simp-all*

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

variables:
 $ms :: \mathbb{Z}\ list$

Missing lemma about *inv-Moves* $((x{::}\mathbb{Z})\ \#\ (xs{::}\mathbb{Z}\ list))$ 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-Move x*
 2. *inv-Moves* $(x \# xs) \Longrightarrow$ *inv-Moves xs*
 3. *inv-Move x* $\wedge$ *inv-Moves xs* $\Longrightarrow$ *inv-Moves* $(x \# xs)$
variables:
  xs :: $\mathbb{Z}$ *list*
  x :: $\mathbb{Z}$

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

oops

declare[[*show-types=false*]]
lemma *l-inv-Moves-Hd*:
  inv-Moves $(x \# xs) \Longrightarrow$ *inv-Move x*
  unfolding *inv-Moves-def*
   by (*simp add*: *l-inv-SeqElems-Cons*)

lemma *l-inv-Moves-Tl*:
  inv-Moves $(x \# xs) \Longrightarrow$ *inv-Moves xs*
unfolding *inv-Moves-def*
apply (*intro conjI*)
apply (*simp*)
apply (*simp add*: *pre-sum-elems-def*)
unfolding *Let-def*
apply (*elim conjE*)
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)
  apply (*simp*, *elim conjE*)
  unfolding *pre-sum-elems-def*
   apply (*simp add*: *l-inv-SeqElems-Cons*)
  apply (*elim conjE*, *intro conjI impI*, *simp-all*)
  unfolding *post-sum-elems-def*
   apply (*simp add*: *l-inv-Moves-Hd*)
   apply (*simp add*: *inv-VDMNat-def l-inv-SeqElems-Cons l-pre-sum-elems l-sum-elems-nat*)
  using *l-inv-Move-nat1 l-inv-SeqElems-Cons* **apply** *fastforce*
  using *l-inv-Move-nat1 l-inv-SeqElems-Cons* **by** *fastforce*

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*
apply (*elim conjE*)
unfolding *inv-Moves-def post-sum-elems-def Let-def*

apply (*elim conjE*, *intro conjI*, *simp-all*)
    apply (*simp add*: *pre-sum-elems-def*)
    apply (*simp add*: *l-inv-SeqElems-Cons*)
  apply (*simp add*: *l-inv-SeqElems-Cons pre-sum-elems-def*)
 using *inv-VDMNat-def l-inv-Move-nat1* **apply** *force*
 nitpick[*user-axioms*=*true*]

Goals not provable when *sum-elems xs* = *MAX-PILE*, because *inv-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-Moves ?ms $\Longrightarrow$ *0* $\leq$ *moves-left ?ms*

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

inv-Moves (*?x* # *?xs*) $\Longrightarrow$ *inv-Move ?x*

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

inv-Moves (*?x* # *?xs*) $\Longrightarrow$ *inv-Moves ?xs*

Proof failures are useful to understand what is wrong:

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

inv-Moves ms $\Longrightarrow$ *0* < *moves-left ms*

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

inv-Moves (*x* # *xs*) = (*inv-Move x* $\wedge$ *inv-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*)

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

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

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

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*) *vdmmod* (*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-left ms*) $-$ *1*) *mod* (*MAX-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-left-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-left ms* $-$ *1*) *mod 4*)
2. $\bigwedge$*ms*. *inv-Moves ms* $\wedge$ *pre-sum-elems ms* $\Longrightarrow$
inv-VDMNat (*moves-left ms*) $\wedge$
inv-VDMNat (*sum-elems ms*) $\wedge$ (*ms* $\neq$ []) $=$ (*0* $<$ *sum-elems ms*)
3. $\bigwedge$*ms*. *inv-Moves ms* $\wedge$ *pre-sum-elems ms* $\Longrightarrow$
(*moves-left ms* $-$ *1*) *mod 4* $\leq$ *moves-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*)
apply (*intro conjI*)
apply (*simp add*: *l-moves-left-nat*)
apply (*simp add*: *pre-sum-elems-def*)
apply (*simp add*: *l-sum-elems-nat*)
apply (*simp add*: *l-pre-sum-elems-sat*)

1. $\bigwedge$*ms*. *inv-Moves ms* $\wedge$ *pre-sum-elems ms* $\Longrightarrow$
(*moves-left ms* $-$ *1*) *mod 4* $\leq$ *moves-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-left ms*)
unfolding *inv-VDMNat-def* **by** (*simp add*: *l-moves-left-nat*)

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*)

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-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*, *intro conjI*)
apply (*simp add*: *pre-sum-elems-def*)
apply (*simp add*: *l-sum-elems-nat*)
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* $\Longrightarrow$ $(ms \neq []) = (0 <$ *sum-elems 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 $(((\textit{moves-left ms}) - 1)$ *mod* $(\textit{MAX-MOV} + 1))$

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*)
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*)
apply (*frule l-sum-elems-nat*)
apply *simp*

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

oops

lemma $0 \leq x \Longrightarrow 0 < a \Longrightarrow 0 < y - (x{::}int) \Longrightarrow 0 < y - (a + x)$

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 $\equiv \forall$ *ms* . *pre-best-move ms* $\longrightarrow$
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 impI*, *elim conjE*,*simp-all*)
apply (*simp add*: *l-inv-VDMNat-moves-left*)
apply (*simp add*: *pre-sum-elems-def*)
apply (*meson inv-Moves-def post-sum-elems-def*)

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

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.

5.13 *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.

5.14 *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* ⇒ *Player*
where
$\textit{flip-current-player } p \equiv (\textit{if } (p = P1) \textit{ then } P2 \textit{ else } P1)$

5.14.1 Specification

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

5.14.2 Satisfiability PO

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

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*))

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

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*

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-Nim ≡ ⦇ *limit* = *MAX-MOV*, *current* = *P1*, *moves* = [] ⦈

6.3 State satisfiability PO

definition
PO-Nim-initialise-sat-obl :: $\mathbb{B}$
where
PO-Nim-initialise-sat-obl ≡ *inv-Nim init-Nim*

theorem *PO-Nim-initialise-sat-obl*
unfolding *PO-Nim-initialise-sat-obl-def inv-Nim-def init-Nim-def inv-Nim-flat-def*
apply *simp*
unfolding *inv-Move-def inv-Moves-def*
apply *auto*
unfolding *pre-fair-play-def post-fair-play-def pre-who-plays-next-def post-who-plays-next-def*
unfolding *pre-sum-elems-def post-sum-elems-def inv-Move-def inv-VDMNat-def inv-VDMNat1-def*
apply *auto*
unfolding *inv-Moves-def isFirst-def inv-Player-def*
apply *auto*
unfolding *pre-sum-elems-def post-sum-elems-def inv-Move-def inv-VDMNat-def inv-VDMNat1-def*
apply *auto*
unfolding *fair-play-def who-plays-next-def len-def*
by *auto*

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 *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)
```

7.1.1 Specification

definition
pre-who-won :: $NimSt \Rightarrow \mathbb{B}$
where
pre-who-won bst $\equiv$ *inv-Nim bst* $\wedge$ *isFirst P1*

definition
post-who-won :: $Player \Rightarrow NimSt \Rightarrow NimSt \Rightarrow \mathbb{B}$
where
post-who-won w bst ast $\equiv$
pre-who-won bst $\longrightarrow$
inv-Player w $\wedge$ *inv-Nim ast* $\wedge$
$(current\ bst) = (current\ ast) \wedge$
$(limit\ bst) = (limit\ ast) \wedge$
$(moves\ bst) = (moves\ ast) \wedge$
pre-who-won-invariant w $(moves\ ast)\ (limit\ ast) \wedge$
post-who-won-invariant w $(moves\ ast)\ (limit\ ast)$ $(who\text{-}won\text{-}invariant\ w\ (moves\ ast)$ $(limit\ ast)) \wedge$
$(who\text{-}won\text{-}invariant\ w\ (moves\ ast)\ (limit\ ast)) \wedge$
$w = current\ ast$

7.1.2 Implementation

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

definition
who-won-complete :: $NimSt \Rightarrow (NimSt \times Player)$
where
who-won-complete bst $\equiv (bst, (current\ bst))$

7.2 *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;
```

7.2.1 Specification

definition
 pre-tally :: *NimSt* ⇒ $\mathbb{B}$
where
 pre-tally bst ≡ *inv-Nim bst* ∧ *pre-who-won bst*

definition
 post-tally :: *NimSt* ⇒ *NimSt* ⇒ $\mathbb{B}$
where
 post-tally bst ast ≡
  pre-tally bst ⟶
  inv-Nim ast ∧ *post-who-won* (*current ast*) *bst ast*

7.2.2 Implementation

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

definition
 tally :: *NimSt* ⇒ *NimSt*
where
 tally bst ≡ (*let p* = *who-won bst in bst*)

7.3 *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);
```

7.3.1 Specification

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

definition
pre-naive-choose-move :: *NimSt* ⇒ $\mathbb{B}$
where
pre-naive-choose-move bst ≡
inv-Nim bst ∧
(*let ms* = (*moves bst*) *in*
pre-moves-left ms ∧
moves-left ms > *0*)

definition
post-naive-choose-move :: *Move* ⇒ *NimSt* ⇒ *NimSt* ⇒ $\mathbb{B}$
where
post-naive-choose-move r bst ast ≡
pre-naive-choose-move bst ⟶
inv-Move r ∧ *inv-Nim ast* ∧
(*let bms* = (*moves bst*) *in*
post-moves-left bms (*moves-left bms*) ∧
r ≤ *moves-left bms*)

7.3.2 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. Proofs involving Hilbert's choice are tricky/difficult.

lemma (*SOME m . m* ∈ {*1* .. *MAX-MOV*}) > *0*
find-theorems *SOME* - . - ∈ - *name*:*Hilbert*
apply (*simp add*: *some-in-eq*)

find-theorems *SOME* - . -
oops
lemma (*SOME m . m* ∈ {*1* .. *MAX-MOV*}) > *0*
apply (*rule someI2*)
by *auto*

lemma (*SOME m . m* ∈ {*1* .. (*3*::*int*)}) > *0*
apply (*rule someI2*)
by *auto*

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

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

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-choose-move0 :: *NimSt* ⇒ *NimSt* × *Move*
where
 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*)))})

7.4 *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 :: 𝔹
where

inv-FIXED-PLAY $\equiv$ *inv-Moves FIXED-PLAY*

7.4.1 Specification

definition
pre-fixed-choose-move :: *NimSt* $\Rightarrow \mathbb{B}$
where
pre-fixed-choose-move bst $\equiv$
pre-naive-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-naive-choose-move RESULT bst ast

7.4.2 Implementation

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

7.5 *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
  ;
```

7.5.1 Specification

definition
pre-first-player-winning-choose-move0 :: *NimSt* $\Rightarrow \mathbb{B}$
where
pre-first-player-winning-choose-move0 bst $\equiv$
inv-Nim bst $\wedge$ *pre-moves-left* (*moves bst*) $\wedge$ *moves-left* (*moves bst*) $>$ *0*

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

definition
post-first-player-winning-choose-move0 :: *Move* ⇒ *NimSt* ⇒ *NimSt* ⇒ 𝔹
where
post-first-player-winning-choose-move0 RESULT bst ast ≡
pre-first-player-winning-choose-move0 bst ⟶
inv-Move RESULT ∧ *inv-Nim ast* ∧
(*let bms* = (*moves bst*);
ams = (*moves ast*);
alim= (*limit ast*);
ac = (*current ast*);
pm = *play-move ac RESULT ams*;
bm = *best-move pm*
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 alim ∧
pre-play-move ac RESULT ams ∧
post-play-move ac RESULT ams pm ∧
pre-best-move pm ∧ *post-best-move pm* (*best-move pm*) ∧

(*limit bst*) = (*limit ast*) ∧ (*current bst*) = (*current ast*) ∧ (*moves bst*) = (*moves ast*)
∧
RESULT < *moves-left ams* ∧
(*will-first-player-win alim* ⟶ *best-move pm* = *0*)
)

definition
post-first-player-winning-choose-move :: *Move* ⇒ *NimSt* ⇒ *NimSt* ⇒ 𝔹
where
post-first-player-winning-choose-move RESULT bst ast ≡
pre-first-player-winning-choose-move bst ⟶
inv-Move RESULT ∧ *inv-Nim ast* ∧
(*let bms* = (*moves bst*);
ams = (*moves ast*);
alim= (*limit ast*);
ac = (*current ast*);
pm = *play-move ac RESULT ams*;

$bm = best\text{-}move\ pm$
in
$pre\text{-}moves\text{-}left\ ams \wedge post\text{-}moves\text{-}left\ ams\ (moves\text{-}left\ ams) \wedge$
$pre\text{-}who\text{-}plays\text{-}next\ ams \wedge post\text{-}who\text{-}plays\text{-}next\ ams\ (who\text{-}plays\text{-}next\ ams) \wedge$
$pre\text{-}will\text{-}first\text{-}player\text{-}win\ alim \wedge$
$pre\text{-}play\text{-}move\ ac\ RESULT\ ams \wedge$
$post\text{-}play\text{-}move\ ac\ RESULT\ ams\ pm \wedge$
$pre\text{-}best\text{-}move\ pm \wedge post\text{-}best\text{-}move\ pm\ (best\text{-}move\ pm) \wedge$
$(limit\ bst) = (limit\ ast) \wedge (current\ bst) = (current\ ast) \wedge (moves\ bst) = (moves\ ast)$
$\wedge$
$RESULT < moves\text{-}left\ ams \wedge$
$(will\text{-}first\text{-}player\text{-}win\ alim \longrightarrow best\text{-}move\ pm = 0)$
$)$

7.5.2 Implementation

definition
first-player-winning-choose-move :: *NimSt* ⇒ *Move*
where
first-player-winning-choose-move st ≡ *best-move* (*moves st*)

7.5.3 Example PO: operation satisfiability

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

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

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-first-player-winning-choose-move-sat-obl ≡
$\forall\ bst\ .\ pre\text{-}first\text{-}player\text{-}winning\text{-}choose\text{-}move\ bst \longrightarrow$
$(\exists\ RESULT\ ast\ .\ post\text{-}first\text{-}player\text{-}winning\text{-}choose\text{-}move\ RESULT\ bst\ ast)$

definition
PO-first-player-winning-choose-move-sat-exp-obl0 :: $\mathbb{B}$
where
PO-first-player-winning-choose-move-sat-exp-obl0 $\equiv$
$\forall$ *bst . pre-first-player-winning-choose-move0 bst* $\longrightarrow$
post-first-player-winning-choose-move0 (*first-player-winning-choose-move bst*) *bst bst*

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

As an illustration, a naive attempt at these kind of proofs by simply expanding definitions and doing layered simplification 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 simp
unfolding inv-Nim-def inv-Nim-flat-def
     inv-Moves-def inv-SeqElems-def
     inv-Move-def max-def
 apply simp
 unfolding
     pre-moves-left-def
     pre-best-move-def
     pre-best-move0-def
     post-best-move-def
     pre-who-plays-next-def
     pre-fair-play-def
     pre-play-move-def
     pre-sum-elems-def
 apply simp
 unfolding  moves-left-def
     best-move-def
     who-plays-next-def
     fair-play-def
     play-move-def
 apply simp
 unfolding  isFirst-def
     will-first-player-win-def
 apply safe
oops
```

7.6 *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
```

7.6.1 Specification

definition

$pre\text{-}save :: Move \Rightarrow NimSt \Rightarrow \mathbb{B}$

where

$pre\text{-}save\ choice\ bst \equiv$
$\quad inv\text{-}Nim\ bst \wedge inv\text{-}Move\ choice \wedge$
$\quad (let\ bc = (current\ bst);$
$\qquad bms = (moves\ bst)$
$\quad in$
$\qquad pre\text{-}play\text{-}move\ bc\ choice\ bms)$

definition

$post\text{-}save :: Move \Rightarrow NimSt \Rightarrow NimSt \Rightarrow \mathbb{B}$

where

$post\text{-}save\ choice\ bst\ ast \equiv$
$\quad inv\text{-}Nim\ bst \wedge inv\text{-}Nim\ ast \wedge inv\text{-}Move\ choice \wedge$
$\quad (let\ bc = (current\ bst);$
$\qquad ac = (current\ ast);$
$\qquad bms = (moves\ bst);$
$\qquad ams = (moves\ ast)\ in$
$\qquad ams = (play\text{-}move\ bc\ choice\ bms) \wedge$
$\qquad post\text{-}play\text{-}move\ bc\ choice\ bms\ ams \wedge$
$\qquad post\text{-}flip\text{-}current\text{-}player\ bc\ (flip\text{-}current\text{-}player\ bc) \wedge$
$\qquad bc \neq ac$
$\quad)$

7.6.2 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* ⇒ *NimSt* ⇒ *NimSt*
where
  save choice bst ≡
   (*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* ⇒ *Move* ⇒ *NimSt*
where
  save0 bst choice ≡
   (*let ms* = *play-move* (*current bst*) *choice* (*moves bst*);
      next = *flip-current-player* (*current bst*) *in*
      bst ⦇ *current* := *next*, *moves* := *ms* ⦈)

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 ≡
   pre-naive-play-game bst ⟶
   inv-Nim ast ∧
   (*let ams* = (*moves ast*) *in moves-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*

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

definition
naive-play-game0 :: *NimSt* ⇒ *NimSt*
where
naive-play-game0 bst ≡
tally(*naive-play-game-loop bst*)

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

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

pre-fixed-play-game bst $\equiv$ *pre-naive-play-game bst*

definition
post-fixed-play-game :: *NimSt* $\Rightarrow$ *NimSt* $\Rightarrow$ $\mathbb{B}$
where
post-fixed-play-game bst ast $\equiv$ *post-naive-play-game bst ast*

7.9.2 Implementation

definition
fixed-play-game-loop :: *NimSt* $\Rightarrow$ *NimSt*
where
fixed-play-game-loop bst $\equiv$
while $(\lambda$ *bst . moves-left* $($*moves bst*$) > 0)$
$(\lambda$ *bst . save* $($*fixed-choose-move bst*$)$ *bst*$)$
bst

definition
fixed-play-game :: *NimSt* $\Rightarrow$ *NimSt*
where
fixed-play-game bst $\equiv$ $($*naive-play-game-loop* ;; *tally*, *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

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

definition
post-first-win-game :: *NimSt* $\Rightarrow$ *NimSt* $\Rightarrow$ $\mathbb{B}$

where
post-first-win-game bst ast $\equiv$ *post-naive-play-game bst ast*

7.10.2 Implementation

definition
first-win-game-loop :: *NimSt* $\Rightarrow$ *NimSt*
where
first-win-game-loop bst $\equiv$
    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* $\Rightarrow$ *NimSt*
where
first-win-game bst $\equiv$ (*first-win-game-loop* ;; *tally*, *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 $\equiv \exists\ m\ .\ inv\text{-}Move\ m \wedge\ m \leq MAX\text{-}MOV$

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

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

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*

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*

apply (*intro allI impI*)
apply *simp*
apply (*erule sum-elems.elims*)
 apply *simp+*
 unfolding *len-def*
 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 applyVDMSeq-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 $\equiv \forall\ ms\ .\ inv\text{-}SeqElems\ inv\text{-}Move\ ms \longrightarrow$
 $post\text{-}sum\text{-}elems\ ms\ (case\ ms\ of\ [] \Rightarrow 0 \mid (x\#xs) \Rightarrow x + sum\text{-}elems\ xs)$

theorem *PO04-sum-elems-post-obl*
unfolding *PO04-sum-elems-post-obl-def inv-Move-def inv-VDMNat1-def inv-VDMNat-def pre-sum-elems-def post-sum-elems-def*
apply (*rule allI*)
apply (*case-tac ms*)
  apply (*intro impI conjI iffI*, *simp-all*)
apply (*subgoal-tac inv-SeqElems inv-Move ms*)
apply (*frule l-sum-elems-nat*)
  apply (*simp add*: *l-sum-elems-nat*)
  using *l-pre-sum-elems* **apply** *force*
nitpick[*user-axioms*]
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

8.5 Proving function and operation satisfiability POs

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:

$\forall\, i.\ \textit{pre-f}\ i \longrightarrow \textit{post-f}\ i\ \textit{expr}$

That is, if the pre condition holds (*i.e.*, *pre-f i*), then 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.

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!**

8.6 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 Satisfiability PO *play-move*

9.1 Simpler variant of *play-move*

A simpler (earlier) version of *play-move* was defined in VDM as:

```
play: Move * Moves -> Moves
play(m, s) == s ^ [m]
pre
  m <= moves_left(s)
  and
  moves_left(s) > 0
post
  sum_elems(s) < sum_elems(RESULT)
  and
  sum_elems(s) + m = sum_elems(RESULT)
```

It is useful here as it is simpler than the current version, which we will prove below. Also, we define the version of *inv-Moves* that doesn't take the specification (pre/post) of *sum-elems* below.

definition
 $\textit{inv-MovesNim0} :: \textit{Moves} \Rightarrow \mathbb{B}$
where

inv-MovesNim0 s $\equiv$
inv-SeqElems inv-Move s $\wedge$
$(sum\text{-}elems\ s) \leq MAX\text{-}PILE \wedge$
$((sum\text{-}elems\ s) = MAX\text{-}PILE \longrightarrow s\ \$\ (len\ s) = 1)$

definition
pre-play-moveNim0 :: $Move \Rightarrow Moves \Rightarrow \mathbb{B}$
where
pre-play-moveNim0 m s $\equiv$
inv-Move m $\wedge$ *inv-MovesNim0 s* $\wedge$
$m \leq (moves\text{-}left\ s) \wedge (moves\text{-}left\ s) > 0$

definition
post-play-moveNim0 :: $Move \Rightarrow Moves \Rightarrow Moves \Rightarrow \mathbb{B}$
where
post-play-moveNim0 m s RESULT $\equiv$
inv-Move m $\wedge$ *inv-MovesNim0 s* $\wedge$ *inv-MovesNim0 RESULT* $\wedge$
$sum\text{-}elems\ s < sum\text{-}elems\ RESULT \wedge$
$sum\text{-}elems\ s + m = sum\text{-}elems\ RESULT$

definition
PO-play-moveNim0-sat-obl0 :: $\mathbb{B}$
where
PO-play-moveNim0-sat-obl0 $\equiv \forall\ m\ s\ .$ *inv-Move m* $\longrightarrow$ *inv-MovesNim0 s* $\longrightarrow$
pre-play-moveNim0 m s $\longrightarrow$ *post-play-moveNim0 m s* $(s\ @\ [m])$

theorem *PO-play-moveNim0-sat-obl0*
using[[*show-types=false*]]
unfolding *PO-play-moveNim0-sat-obl0-def*
apply *simp*
unfolding
pre-play-moveNim0-def post-play-moveNim0-def
apply *simp*
unfolding *inv-Moves-def inv-MovesNim0-def*
apply *simp*
apply *safe*

too many similar goals. expanding won't work.

oops

9.2 PO for the current version of *play-move*

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*$)$

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

using[[*show-types=false*]]
unfolding *PO-play-move-sat-obl-def*
apply *simp*
unfolding
pre-play-move-def post-play-move-def
apply *simp*
apply (*intro allI impI conjI*,*elim conjE*)
apply (*rule-tac x=s* @ [*m*] **in** *exI*)
apply (*simp*)
apply *safe*

1. $\bigwedge p\ m\ s.$
$[\![$*inv-Move m*; *inv-Moves s*; *post-fair-play p s True*; *inv-Player p*;
pre-moves-left s; *pre-fair-play p s*; *fair-play p s*;
moves-left s $= 1$; *moves-left s* $\neq m$; $\neg$ *False*$]\!]$
$\Longrightarrow$ *inv-Moves* (*s* @ [*m*])
2. $\bigwedge p\ m\ s.$
$[\![$*inv-Move m*; *inv-Moves s*; *post-fair-play p s True*; *inv-Player p*;
pre-moves-left s; *pre-fair-play p s*; *fair-play p s*;
moves-left s $= 1$; *moves-left s* $\neq m$; $\neg$ *False*$]\!]$
$\Longrightarrow$ *pre-sum-elems s*
3. $\bigwedge p\ m\ s.$
$[\![$*inv-Move m*; *inv-Moves s*; *post-fair-play p s True*; *inv-Player p*;
pre-moves-left s; *pre-fair-play p s*; *fair-play p s*;
moves-left s $= 1$; *moves-left s* $\neq m$; $\neg$ *False*$]\!]$
$\Longrightarrow$ *pre-sum-elems* (*s* @ [*m*])
4. $\bigwedge p\ m\ s.$
$[\![$*inv-Move m*; *inv-Moves s*; *post-fair-play p s True*; *inv-Player p*;
pre-moves-left s; *pre-fair-play p s*; *fair-play p s*;
moves-left s $= 1$; *moves-left s* $\neq m$; $\neg$ *False*$]\!]$
$\Longrightarrow$ *post-sum-elems s* (*sum-elems s*)
5. $\bigwedge p\ m\ s.$
$[\![$*inv-Move m*; *inv-Moves s*; *post-fair-play p s True*; *inv-Player p*;
pre-moves-left s; *pre-fair-play p s*; *fair-play p s*;
moves-left s $= 1$; *moves-left s* $\neq m$; $\neg$ *False*$]\!]$
$\Longrightarrow$ *post-sum-elems* (*s* @ [*m*]) (*sum-elems* (*s* @ [*m*]))
6. $\bigwedge p\ m\ s.$
$[\![$*inv-Move m*; *inv-Moves s*; *post-fair-play p s True*; *inv-Player p*;
pre-moves-left s; *pre-fair-play p s*; *fair-play p s*;
moves-left s $= 1$; *moves-left s* $\neq m$; $\neg$ *False*$]\!]$
$\Longrightarrow$ *sum-elems s* $<$ *sum-elems* (*s* @ [*m*])
7. $\bigwedge p\ m\ s.$
$[\![$*inv-Move m*; *inv-Moves s*; *post-fair-play p s True*; *inv-Player p*;
pre-moves-left s; *pre-fair-play p s*; *fair-play p s*;
moves-left s $= 1$; *moves-left s* $\neq m$; $\neg$ *False*$]\!]$
$\Longrightarrow$ *sum-elems s* $+ m =$ *sum-elems* (*s* @ [*m*])
8. $\bigwedge p\ m\ s.$
$[\![$*inv-Move m*; *inv-Moves s*; *post-fair-play p s True*; *inv-Player p*;
pre-moves-left s; *pre-fair-play p s*; *fair-play p s*;

moves-left s $=$ *1*; *moves-left s* $\neq$ *m*; $\neg$ *False*; *fair-play p* $(s \mathbin{@} [m])$$]\!]$
$\Longrightarrow$ *False*

9. $\bigwedge$*p m s.*
$[\![$*inv-Move 1*; *inv-Moves s*; *post-fair-play p s True*; *inv-Player p*;
pre-moves-left s; *pre-fair-play p s*; *fair-play p s*;
moves-left s $=$ *1*; $\neg$ *False*$]\!]$
$\Longrightarrow$ *inv-Moves* $(s \mathbin{@} [1])$

10. $\bigwedge$*p m s.*
$[\![$*inv-Move 1*; *inv-Moves s*; *post-fair-play p s True*; *inv-Player p*;
pre-moves-left s; *pre-fair-play p s*; *fair-play p s*;
moves-left s $=$ *1*; $\neg$ *False*$]\!]$
$\Longrightarrow$ *pre-sum-elems s*

A total of 32 subgoals...

These goals will require various lemmas.

oops

lemma *inv-Move m* $\Longrightarrow$
inv-Moves s $\Longrightarrow$ *inv-Moves* $(s \mathbin{@} [m])$
unfolding *inv-Moves-def*
apply *safe*

important one that will be difficult to finish

oops

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

9.3 Naive attempt with split lemmas

declare [[*show-types=false*]]
theorem *PO-play-move-sat-exp-obl*
unfolding *PO-play-move-sat-exp-obl-def*
apply *safe*
unfolding *post-play-move-def*
apply *safe*
unfolding *post-sum-elems-def*
unfolding *pre-play-move-def pre-sum-elems-def*
apply *simp-all*

too many subgoals if you apply safe

apply *safe*
oops

Lemmas based on the goals before applying safe above.

lemma *l1*: *inv-Move m* $\Longrightarrow$ *inv-Moves s* $\Longrightarrow$ *pre-play-move0 p m s* $\Longrightarrow$ *inv-Moves* (*play-move p m s*)
unfolding *play-move-def pre-play-move0-def*
apply (*safe*, *simp-all*)

sledgehammer failed

oops

lemma *l1*: *inv-Move m* $\Longrightarrow$ *inv-Moves s* $\Longrightarrow$ *pre-play-move0 p m s* $\Longrightarrow$ *inv-Moves* (*play-move p m s*)
unfolding *play-move-def pre-play-move0-def*
apply (*safe*, *simp-all*)
unfolding *moves-left-def inv-Moves-def Let-def*
apply (*simp*, *safe*, *simp-all*)
unfolding *post-sum-elems-def pre-sum-elems-def*

naive strategy doesn't work. You can use sorry to discover the splitting lemmas to be proved next: that is, will they help the larger proof?

oops

lemma *l2*: *inv-Move m* $\Longrightarrow$ *inv-Moves s* $\Longrightarrow$ *pre-play-move0 p m s* $\Longrightarrow$ *pre-sum-elems s*
using *inv-Moves-def* **by** *blast*

lemma *l3*: *inv-Move m* $\Longrightarrow$ *inv-Moves s* $\Longrightarrow$ *pre-play-move0 p m s* $\Longrightarrow$ *pre-sum-elems* (*play-move p m s*)
oops

The fact this proof is the same as l2, might mean they are the same goal?

lemma *l4*: *inv-Move m* $\Longrightarrow$ *inv-Moves s* $\Longrightarrow$ *pre-play-move0 p m s* $\Longrightarrow$ *inv-SeqElems inv-Move s*
using *inv-Moves-def* **by** *blast*

lemma *l2-same-l4*: *inv-Move m* $\Longrightarrow$ *inv-Moves s* $\Longrightarrow$ *pre-play-move0 p m s* $\Longrightarrow$ *pre-sum-elems s*
unfolding *pre-sum-elems-def* **by** (*simp add*: *l4*)

Study *l-sum-elems-nat x*: the meson proof

lemma *l5*: *inv-Move m* $\Longrightarrow$ *inv-Moves s* $\Longrightarrow$ *pre-play-move0 p m s* $\Longrightarrow$ *inv-VDMNat* (*sum-elems s*)

by (*meson inv-Moves-def post-sum-elems-def*)

l5 seems the same as *l6*

lemma *l6*: *inv-Move m* $\Longrightarrow$ *inv-Moves s* $\Longrightarrow$ *pre-play-move0 p m s* $\Longrightarrow$ *s* $\neq$ [] $\Longrightarrow$ *0* $<$ *sum-elems s*
by (*meson inv-Moves-def post-sum-elems-def*)

l7 seems similar / more general to *l3*

lemma *l7*: *inv-Move m* $\Longrightarrow$
inv-Moves s $\Longrightarrow$ *pre-play-move0 p m s* $\Longrightarrow$ *inv-SeqElems inv-Move* (*play-move p m s*)

oops

lemma *l8*: *inv-Move m* $\Longrightarrow$
inv-Moves s $\Longrightarrow$ *pre-play-move0 p m s* $\Longrightarrow$ *inv-VDMNat* (*sum-elems* (*play-move p m s*))

oops

l9 seems similar to *l5*

lemma *l9*: *inv-Move m* $\Longrightarrow$
inv-Moves s $\Longrightarrow$
pre-play-move0 p m s $\Longrightarrow$ *play-move p m s* $\neq$ [] $\Longrightarrow$ *0* $<$ *sum-elems* (*play-move p m s*)

oops

lemma *l10*: *inv-Move m* $\Longrightarrow$
inv-Moves s $\Longrightarrow$ *pre-play-move0 p m s* $\Longrightarrow$ *sum-elems s* $<$ *sum-elems* (*play-move p m s*)
unfolding *pre-play-move0-def play-move-def*
apply (*simp*)
apply (*safe*,*simp-all*)
apply (*induct s*)
apply *simp-all*
oops

lemma *l11*: *inv-Move m* $\Longrightarrow$
inv-Moves s $\Longrightarrow$ *pre-play-move0 p m s* $\Longrightarrow$ *sum-elems s* $+$ *m* $=$ *sum-elems* (*play-move p m s*)
unfolding *pre-play-move0-def play-move-def*
apply (*simp*)
apply (*safe*,*simp-all*) **oops**

lemma *l12*: *inv-Move m* $\Longrightarrow$ *inv-Moves s* $\Longrightarrow$ *fair-play p s* $\Longrightarrow$ $\neg$ *fair-play p* (*play-move p m s*)
unfolding *fair-play-def who-plays-next-def play-move-def*
apply (*simp split*: *if-splits*)
apply *auto*[*1*]
by (*simp add*: *mod-add-cong*)

12 lemmas, 5 lemmas (42l11 is close to l10, other unproved lemmas very much depend on those.

Lemmas with "sorry" are dangerous: if you don't prove them, you haven't finished. I left the above lemmas in place to enable you to see how they play in the proof below (i.e. change oops for sorry to see).

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

unfolding *PO-play-move-sat-exp-obl-def*
unfolding *post-play-move-def*
unfolding *post-sum-elems-def*
apply (*safe*, *simp-all*)+

All goals below are discovered with sledgehammer.

defer
apply (*simp add*: *pre-moves-left-def pre-play-move-def*)
apply (*simp add*: *inv-Moves-def l-inv-SeqElems-append play-move-def pre-sum-elems-def*)
apply (*simp add*: *inv-Moves-def pre-play-move-def*)
using *inv-VDMNat-def l-sum-elems-nat pre-sum-elems-def* **apply** *blast*
apply (*simp add*: *l-pre-sum-elems-sat*)
apply (*simp add*: *inv-VDMNat-def l-sum-elems-nat pre-sum-elems-def*)
defer
defer
defer
using *l12 pre-play-move-def*
apply (*simp add*: *play-move-def*)
oops

Only l1 l10 l11 l12 are needed, but first three have "sorried" proofs. We need to finish their proofs for this proof to be valid: at least we established that they are the lemmas that will help with this proof.

I leave their proof as an exercise — some of it will be redone/reorganised below anyhow.

9.4 Lemmas about auxiliary function *sum-elems*

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

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

lemma *l-concat-append* : *nconcat xs ys* = *xs* @ *ys*
by (*induct xs*, *simp-all*)

Definitions using sequence cons (*x* # *xs*) will need lemmas about sequence append (*s* @ *t*).

lemma *l-sum-elems-nconcat*: *sum-elems* (*nconcat ms* [*m*]) = (*m* + *sum-elems ms*)

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

Some interesting lemmas about *sum-elems*

inv-SeqElems inv-Move ?s $\Longrightarrow$ *0* $\leq$ *sum-elems ?s*

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

9.5 Lemma discovery through failed proof attempts

The proof attempt above for succeeded but there were lemmas missing their proofs. Let's try again, this time with the current version of *play-move* and without any "sorry" theorems.

1 Naive attempt: layered expansion followed by simplification.

theorem *PO-play-move-sat-exp-obl*
unfolding *PO-play-move-sat-exp-obl-def post-play-move-def play-move-def*
apply (*safe*)

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

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

$\textit{post-play-move}\ p\ m\ ms\ (ms\ @\ [m]) \equiv$
$\textit{pre-play-move}\ p\ m\ ms \longrightarrow$
$\textit{inv-Moves}\ (ms\ @\ [m]) \wedge$
$\textit{pre-sum-elems}\ ms \wedge$
$\textit{pre-sum-elems}\ (ms\ @\ [m]) \wedge$
$\textit{post-sum-elems}\ ms\ (\textit{sum-elems}\ ms) \wedge$
$\textit{post-sum-elems}\ (ms\ @\ [m])\ (\textit{sum-elems}\ (ms\ @\ [m])) \wedge$

sum-elems ms $<$ *sum-elems* (*ms* @ [*m*]) $\wedge$
sum-elems ms $+$ *m* $=$ *sum-elems* (*ms* @ [*m*]) $\wedge$
$\neg$ *fair-play p* (*ms* @ [*m*]) $\wedge$ *ms* $\sqsubseteq$ *ms* @ [*m*]

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

apply (*simp-all*)

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

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.

What each say:

1. *inv-Moves* is preserved on *s* @ [*m*]
2. *pre-sum-elems s* is trivial from *pre-play-move*
3.
4. *post-sum-elems s* is trivial from *pre-play-move*
5.

oops

9.5.1 Lemmas per subgoal

For each subgoal above, let's try and create lemmas (and their generalisations). The first subgoal is difficult: it relies on *inv-Moves*, which contains various predicates, so we start the next goal.

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*)

lemma *l-pre-sume-seqelems-move*: *inv-SeqElems inv-Move ms* $\Longrightarrow$ *pre-sum-elems ms*

by (*simp add*: *pre-sum-elems-def*)

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*

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

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

Groups of definitions can be named to make their unfolding in one go.

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*

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*)

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*)

This is a variation over $[\![$*inv-Move ?m*; *inv-Moves ?s*; *fair-play ?p ?s*$]\!] \Longrightarrow \neg$ *fair-play ?p* (*play-move ?p ?m ?s*).

lemma *l-sg6-2-fair-play*:
fair-play p s $\Longrightarrow \neg$ *fair-play p* (*s* @ [*m*])
unfolding *fair-play-def who-plays-next-def*
apply (*safe*,*simp split*: *if-splits*)
unfolding *len-def*
by *presburger*+

lemma *l-sg6-not-fair-play-play-move*:
inv-Move m $\Longrightarrow$
inv-Moves s $\Longrightarrow$ *pre-play-move p m s* $\Longrightarrow$ $\neg$ *fair-play p* $(s \mathbin{@} [m])$
unfolding *pre-play-move-def*
by (*simp add*: *l-sg6-2-fair-play*)

9.5.2 General lemmas are easier

The actual VDM (declared) postcondition represents some of the subgoals above. 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* $(ms \mathbin{@} [m]) = (m + \textit{sum-elems } ms)$
by (*induct ms*, *simp-all*)

lemma *l-sum-elems-append-gen*: *sum-elems* $(s \mathbin{@} t) = (\textit{sum-elems } s + \textit{sum-elems } t)$
by (*induct s*, *simp-all*)

Similarly, this exercise suggested the introduction of various other lemmas for definitions in `VDMToolkit.thy`, such as:

$?s \neq [] \Longrightarrow 0 < \textit{len } ?s$
$\textit{len } ?s \leq \textit{len } (?s \mathbin{@} ?t)$
$\textit{len } (?x \# ?xs) = 1 + \textit{len } ?xs$

$\textit{elems } (?xs \mathbin{@} [?x]) = \textit{insert } ?x \ (\textit{elems } ?xs)$
$\textit{elems } (?x \# ?xs) = \textit{insert } ?x \ (\textit{elems } ?xs)$

$\textit{inv-SeqElems } ?f \ (?xs \mathbin{@} [?x]) = (?f \ ?x \wedge \textit{inv-SeqElems } ?f \ ?xs)$
$\textit{inv-SeqElems } ?f \ (?a \# ?s) = (?f \ ?a \wedge \textit{inv-SeqElems } ?f \ ?s)$

$[\![?s \neq [];\ \textit{inv-SeqElems } (\lambda x.\ x \neq \textit{undefined}) \ ?s]\!] \Longrightarrow ?s \ \$ \ \textit{len } ?s \neq \textit{undefined}$
$(?ms \mathbin{@} [?m]) \ \$ \ \textit{len } (?ms \mathbin{@} [?m]) = ?m$
$(?m \# ?ms) \ \$ \ \textit{len } (?m \# ?ms) = (\textit{if } ?ms = [] \textit{ then } ?m \textit{ else } ?ms \ \$ \ \textit{len } ?ms)$

$\textit{inds } (?xs \mathbin{@} [?x]) = \textit{insert } (\textit{len } (?xs \mathbin{@} [?x])) \ (\textit{inds } ?xs)$
$?s \neq [] \Longrightarrow \textit{len } ?s \in \textit{inds } ?s$

9.6 "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-obl*

...

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

Goal about *inv-Moves* $(s\ @\ [m])$ is missing above; postpone it for now.

defer
apply (*simp add*: *l-sg2-pre-sume pre-play-move-def*)
apply (*simp add*: *l-sg3-pre-sume-append pre-play-move-def*)
apply (*meson inv-Moves-def pre-play-move-def*)
apply (*simp add*: *l-sg5-post-sume-append pre-play-move-def*)
apply (*simp add*: *l-inv-Move-nat1 l-sum-elems-append pre-play-move-def*)
apply (*simp add*: *l-sum-elems-append*)
apply (*simp add*: *l-sg6-2-fair-play pre-play-move-def*)

1. $\bigwedge p\ m\ s.\ \textit{pre-play-move}\ p\ m\ s \Longrightarrow \textit{inv-Moves}\ (s\ @\ [m])$

Yes! So, for the difficult case: it generates more subgoals :-(. Will avoid safe here, but otherwise would have to deal with the many it generates.

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

1. $\bigwedge p\ m\ s.\ \textit{pre-play-move}\ p\ m\ s \Longrightarrow \textit{inv-SeqElems}\ \textit{inv-Move}\ (s\ @\ [m])$
2. $\bigwedge p\ m\ s.\ \textit{pre-play-move}\ p\ m\ s \Longrightarrow \textit{pre-sum-elems}\ (s\ @\ [m])$
3. $\bigwedge p\ m\ s.$
   $\textit{pre-play-move}\ p\ m\ s \Longrightarrow \textit{post-sum-elems}\ (s\ @\ [m])\ (\textit{sum-elems}\ (s\ @\ [m]))$
4. $\bigwedge p\ m\ s.\ \textit{pre-play-move}\ p\ m\ s \Longrightarrow \textit{sum-elems}\ (s\ @\ [m]) \le \textit{MAX-PILE}$
5. $\bigwedge p\ m\ s.$
   $[\![\textit{pre-play-move}\ p\ m\ s;\ \textit{sum-elems}\ (s\ @\ [m]) = \textit{MAX-PILE}]\!]$
   $\Longrightarrow (s\ @\ [m])\ !\ \textit{nat}\ (\textit{len}\ s) = \textit{1}$

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

$\textit{inv-Moves}\ ?s \equiv$
$\textit{inv-SeqElems}\ \textit{inv-Move}\ ?s\ \wedge$
$\textit{pre-sum-elems}\ ?s\ \wedge$
$(\textit{let}\ r = \textit{sum-elems}\ ?s$
$\ \textit{in}\ \textit{post-sum-elems}\ ?s\ r \wedge r \le \textit{MAX-PILE} \wedge (r = \textit{MAX-PILE} \longrightarrow ?s\ \$\ \textit{len}\ ?s = \textit{1}))$

oops

lemma *inv-Move m* $\Longrightarrow$ *inv-Moves s* $\Longrightarrow$ *inv-Moves* (*s* @ [*m*])
unfolding *inv-Moves-defs Let-def*
apply (*simp*,*safe*)
using *inv-VDMNat-def l-inv-Move-nat1 l-sum-elems-append* **apply** *force*
using *l-inv-Move-nat1 l-sum-elems-append* **apply** *fastforce*
defer defer
apply (*simp add*: *inv-VDMNat-def l-inv-Move-nat1 le-less*)
apply (*simp add*: *l-inv-Move-nat1*)
apply (*simp add*: *inv-Move-def*)
apply (*simp add*: *inv-Move-def*)
using *inv-VDMNat-def l-inv-Move-nat1 l-sum-elems-append* **apply** *force*
using *l-inv-Move-nat1 l-sum-elems-append* **apply** *fastforce* **defer defer**
apply (*simp add*: *inv-VDMNat-def l-inv-Move-nat1 le-less*)
apply (*simp add*: *l-inv-Move-nat1*)
apply (*simp add*: *inv-Move-def*)
apply (*simp add*: *inv-Move-def*)
oops

Alternative variant proof slightly simpler to the same outcome of goal on *inv-Moves* missing.

theorem *PO-play-move-sat-exp-obl*
unfolding *PO-play-move-sat-exp-obl-def post-play-move-def play-move-def*
apply (*simp*,*safe*)

defer
apply (*simp add*: *l-sg2-pre-sume pre-play-move-def*)
apply (*simp add*: *l-sg3-pre-sume-append pre-play-move-def*)
apply (*meson inv-Moves-def pre-play-move-def*)
apply (*simp add*: *l-sg5-post-sume-append pre-play-move-def*)
apply (*simp add*: *l-inv-Move-nat1 l-sum-elems-append pre-play-move-def*)
apply (*simp add*: *l-sum-elems-append*)
apply (*simp add*: *l-sg6-2-fair-play pre-play-move-def*)
oops

9.6.1 Handling (last?) difficult case on *inv-Moves* (*s* @ [*m*])

For the final case, we start with the naive attempt from remaining goal as

lemma *l-sg1-inv-Moves-append*: *inv-Move m* $\Longrightarrow$ *inv-Moves s* $\Longrightarrow$ *pre-play-move p m s* $\Longrightarrow$ *inv-Moves* (*s* @ [*m*])
unfolding *inv-Moves-def Let-def*
apply (*simp*,*safe*)

This generates (10) new subgoals (some somewhat repeated), which some Sledgehammer already finds proofs for

apply (*simp add*: *l-inv-SeqElems-append*)
apply (*simp add*: *l-inv-SeqElems-append pre-sum-elems-def*)
apply (*metis l-inv-SeqElems-append l-sg4-post-sume l-sum-elems-append*)
defer

using *l-applyVDMSeq-append-last l-sum-elems-append moves-left-def pre-play-move-def*
apply *force*
apply (*simp add*: *l-inv-SeqElems-append*)
apply (*simp add*: *l-inv-SeqElems-append pre-sum-elems-def*)
apply (*simp add*: *inv-Moves-def l-sg5-post-sume-append*)
defer
using *l-applyVDMSeq-append-last l-sum-elems-append moves-left-def pre-play-move-def*
apply *force*

Remaining (2) subgoals are about *sum-elems* (s @ $[m]$) being within *MAX-PILE*

1. $[\![$*inv-Move m*; *pre-play-move p m s*; *inv-SeqElems inv-Move s*;
 pre-sum-elems s; *post-sum-elems s* (*sum-elems s*);
 sum-elems s $\leq$ *MAX-PILE*; *sum-elems s* $\neq$ *MAX-PILE*$]\!]$
 $\Longrightarrow$ *sum-elems* (s @ $[m]$) $\leq$ *MAX-PILE*
2. $[\![$*inv-Move m*; *pre-play-move p m s*; *inv-SeqElems inv-Move s*;
 pre-sum-elems s; *post-sum-elems s* (*sum-elems s*);
 sum-elems s $\leq$ *MAX-PILE*; *s* \$ *len s* = *1*$]\!]$
 $\Longrightarrow$ *sum-elems* (s @ $[m]$) $\leq$ *MAX-PILE*

These cases have to do with normal and final play.

oops

lemma *l-sg1-inv-Moves-append*: *pre-play-move p m s* $\Longrightarrow$ *inv-Moves* (s @ $[m]$)
unfolding *inv-Moves-def Let-def*
apply (*simp*,*safe*)
apply (*simp add*: *l-inv-Moves-inv-SeqElems l-inv-SeqElems-append pre-play-move-def*)
apply (*simp add*: *l-sg3-pre-sume-append pre-play-move-def*)
using *l-sg5-post-sume-append pre-play-move-def* **apply** *blast*
defer
apply (*simp add*: *VDMSeq-defs*(*5*) *l-sum-elems-append moves-left-def pre-play-move-def*)

Last goal of interest, but when you open *pre-play-move*

unfolding *pre-play-move-def*
apply (*safe*,*simp-all*)
oops

The remaining goals show how we are getting close. They also reveal that conditionals (through if-then-else or implication lead to case analysis (i.e. more subgoals).

The choice among the sledgehammer-discovered proofs was on the minimal number of lemmas to use. This is useful to remove any cluttering lemmas at the end, something important to help sledgehammer in finding proofs later on.

lemma *l-sg1-4-inv-Moves-maxpile-sume-append*:
inv-Move m $\Longrightarrow$ *inv-SeqElems inv-Move s* $\Longrightarrow$ *sum-elems* (*ms* @ $[m]$) $\leq$ *MAX-PILE*

apply (*simp add*: *l-sum-elems-append*)
apply (*induct ms*)

apply (*simp add*: *inv-Move-def*)
apply *simp*
nitpick

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

We are stuck. Sledgehammer finds nothing, and nitpick finds a potential (but not certain) counterexample. Seems like we need more assumptions. Let us try the last subgoal.

oops

We are really narrowing it down. Let us set them up with extra assumptions from *pre-play-move*.

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

1. $[\![$*inv-Player p*; *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$]\!]$
 $\Longrightarrow$ *moves-left ms* $=$ *m*
2. $[\![$*inv-Player p*; *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; *m* $=$ *1*$]\!]$
 $\Longrightarrow$ *moves-left ms* $\neq$ *1*
3. $[\![$*inv-Player p*; *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; *m* $<$ *moves-left ms*$]\!]$
 $\Longrightarrow$ *moves-left ms* $=$ *m*
4. $[\![$*inv-Player p*; *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; *m* $<$ *moves-left ms*; *m* $=$ *1*$]\!]$
 $\Longrightarrow$ *sum-elems* (*ms* @ [*m*]) $\leq$ *MAX-PILE*

defer
apply (*simp add*: *l-sum-elems-append moves-left-def*)
defer
defer
apply (*simp add*: *l-sum-elems-append moves-left-def*)

Still subgoals we can't easily discover. Try again each as a sub lemma (for third iteration time!)

1. $[\![$*inv-Player p*; *inv-Move m*; *inv-Moves ms*; *pre-moves-left ms*;
 pre-fair-play p ms; *post-fair-play p ms* (*fair-play p ms*);

$\quad$ *fair-play p ms*$]\!]$
$\quad \Longrightarrow$ *moves-left ms* $=$ *m*
2. $[\![$*inv-Player p*; *inv-Move 1*; *inv-Moves ms*; *pre-moves-left ms*;
pre-fair-play p ms; *post-fair-play p ms True*; *fair-play p ms*; *m* $=$ *1*$]\!]$
$\Longrightarrow$ *sum-elems ms* $\neq$ *19*
3. $[\![$*inv-Player p*; *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; *m* $<$ *moves-left ms*$]\!]$
$\Longrightarrow$ *moves-left ms* $=$ *m*

oops

Example lemmas that turn out to be superfluous are commented out in the code.

9.6.2 Generalisation of terms

Understanding more general principle independent of given terms is an important step in proof. We have to prove that *sum-elems* (*s* @ [*m*]) $\leq$ *MAX-PILE* when *sum-elems s* $\leq$ *MAX-PILE* for two cases: 1) normal play (*sum-elems s* $\neq$ *MAX-PILE*) and 2) final play (*s* \$ *len s* $=$ (*1*::'*a*)).

Below we try to generalise it away

lemma *0* $<$ *m* $\Longrightarrow$ *m* $\leq$ *G-MAX-MOV* $\Longrightarrow$ *inv-Moves ms* $\Longrightarrow$ *0* $<$ *G-MAX-PILE* $-$ *sum-elems ms* $\Longrightarrow$ *G-MAX-PILE* $-$ *sum-elems ms* $\neq$ *1*
apply (*rule notI*,*simp*) **oops**

lemma *x* $\geq$ (*0*::*nat*) $\Longrightarrow$ *x* $\leq$ *nat MAX-MOV* $\Longrightarrow$ *list-all* (λ *e* . *e* $\geq$ (*0*::*nat*)) *xs* $\Longrightarrow$
listsum xs $\geq$ *0* $\Longrightarrow$ *listsum xs* $\leq$ *nat MAX-PILE* $\Longrightarrow$ *x* $+$ *listsum xs* $\leq$ *nat MAX-PILE*
apply (*induct x rule*:*nat-induct*)
apply *simp-all*
apply (*induct xs rule*:*list*.*induct*)
apply *simp-all*
defer
apply (*subgoal-tac* ($\bigwedge$*n*. *n* $+$ *listsum x2* $\leq$ *20* $\Longrightarrow$ *n* $\leq$ *2*), *simp*)
apply (*subgoal-tac* ($\forall$ *m* . *m* $+$ *listsum x2* $\leq$ *19*))
apply (*erule-tac x*=*n*+*x1* **in** *allE*, *simp*)
apply (*subgoal-tac* ($\forall$ *n* . *n* $+$ *listsum x2* $\leq$ *20*))
apply *simp-all*
apply (*subgoal-tac* *n* $+$ *listsum x2* $\leq$ *19*)
apply *simp-all*
apply *auto*
oops

lemma *x* $\geq$ (*0*::*nat*) $\Longrightarrow$ *x* $\leq$ *nat MAX-MOV* $\Longrightarrow$ *list-all* (λ *e* . *e* $\geq$ (*0*::*nat*)) *xs* $\Longrightarrow$
listsum xs $\geq$ *0* $\Longrightarrow$ *listsum xs* $\leq$ *nat MAX-PILE* $\Longrightarrow$ *x* $+$ *listsum xs* $\leq$ *nat MAX-PILE*
apply (*induct x rule*:*nat-induct*)
apply *simp-all*
apply (*simp only*: *le-less*)

apply $(erule\ disjE)$
apply $(rule\ disjI1)$
nitpick[*user-axioms*]
oops

lemma $x \leq nat\ MAX\text{-}MOV \Longrightarrow y \leq nat\ MAX\text{-}PILE \Longrightarrow x + y \leq nat\ MAX\text{-}PILE$
nitpick[*user-axioms*]
apply $(induct\ x\ rule{:}nat\text{-}induct)$
apply *simp-all*
apply $(simp\ only{:}\ le\text{-}less)$
apply $(erule\ disjE)$
apply $(rule\ disjI1)$
nitpick[*user-axioms*]
oops

To try and understand what is the problem, we generalise the expressions to simpler terms. It is useful to discover what we know about the operators involved (e.g. $sum\text{-}elems\ (s\ @\ t)$).

find-theorems *sum-elems - name*:*Nim*
find-theorems *sum-elems* (- @ -)

And get to the following unprovable conjecture, which gives the hint to the missing condition of interest, and its improved (provable) version.

lemma *l-sg1-4-1-explore*: $x \leq\ MAX\text{-}MOV \Longrightarrow y \leq\ MAX\text{-}PILE \Longrightarrow y \neq MAX\text{-}PILE \Longrightarrow x + y \leq\ MAX\text{-}PILE$
nitpick
oops

This shows the missing relationship that x (or m) has to have with y (or *sum-elems s*) in order to prove the goal. This needs to come as an assumption from somewhere.

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

9.6.3 Proving the missing cases for $inv\text{-}Moves\ (s\ @\ [m])$ subgoal

More lemmas that turn out to be superfluous are commented out in the code.

The proof commented below shows the dangers of sorried lemmas: it used l1, which wasn't prooved.

lemma *l-sg1-1-missing-assumption*:
$pre\text{-}play\text{-}move\ p\ m\ s \Longrightarrow sum\text{-}elems\ s \neq MAX\text{-}PILE \Longrightarrow m + (sum\text{-}elems\ s) < MAX\text{-}PILE$
unfolding *pre-play-move-def*
apply $(safe,simp\text{-}all)$

Oh man, it generates yet four more subgoals. Let's try avoiding safe

1. $[\![sum\text{-}elems\ s \neq MAX\text{-}PILE;\ inv\text{-}Player\ p;\ inv\text{-}Move\ m;\ inv\text{-}Moves\ s;$

pre-moves-left s; *pre-fair-play p s*; *post-fair-play p s True*;
fair-play p s; $\neg$ *m* + *sum-elems s* < *MAX-PILE*; *moves-left s* = *1*; *m* $\neq$ *1*⟧
$\Longrightarrow$ *False*

2. ⟦*sum-elems s* $\neq$ *MAX-PILE*; *inv-Player p*; *inv-Move 1*; *inv-Moves s*;
pre-moves-left s; *pre-fair-play p s*; *post-fair-play p s True*;
fair-play p s; $\neg$ *sum-elems s* < *19*; *moves-left s* = *1*; *m* = *1*⟧
$\Longrightarrow$ *False*
3. ⟦*sum-elems s* $\neq$ *MAX-PILE*; *inv-Player p*; *inv-Move m*; *inv-Moves s*;
pre-moves-left s; *pre-fair-play p s*; *post-fair-play p s True*;
fair-play p s; *m* < *moves-left s*⟧
$\Longrightarrow$ *m* + *sum-elems s* < *MAX-PILE*
4. ⟦*sum-elems s* $\neq$ *MAX-PILE*; *inv-Player p*; *inv-Move 1*; *inv-Moves s*;
pre-moves-left s; *pre-fair-play p s*; *post-fair-play p s True*;
fair-play p s; *1* < *moves-left s*; *m* = *1*⟧
$\Longrightarrow$ *sum-elems s* < *19*

oops

lemma *l-sg1-1-missing-assumption*:
pre-play-move p m s $\Longrightarrow$ *sum-elems s* $\neq$ *MAX-PILE* $\Longrightarrow$ *m* + (*sum-elems s*) < *MAX-PILE*
unfolding *pre-play-move-def pre-moves-left-def moves-left-def*
apply (*simp*)
unfolding *pre-fair-play-def post-fair-play-def pre-who-plays-next-def post-who-plays-next-def*
apply *simp*
unfolding *pre-sum-elems-def inv-Moves-def post-sum-elems-def*
apply *simp*
unfolding *isFirst-def inv-Player-def inv-Move-def inv-SeqElems-def inv-VDMNat1-def Let-def inv-VDMNat-def*
apply (*elim conjE*, *simp*)
apply *safe*

Oh man, it generates yet four more subgoals. Let's try avoiding safe

oops

lemma *l-sg1-1-missing-assumption*:
pre-play-move p m s $\Longrightarrow$ *sum-elems s* $\neq$ *MAX-PILE* $\Longrightarrow$ *m* + (*sum-elems s*) < *MAX-PILE*
unfolding *pre-play-move-def moves-left-def pre-moves-left-def*
apply (*simp*)
unfolding *inv-Moves-def Let-def*
apply *simp*
apply (*case-tac sum-elems s* $\neq$ *19*)
apply (*simp-all*)
unfolding *post-sum-elems-def*
unfolding *inv-Move-def inv-VDMNat1-def*
apply *simp*
apply (*elim conjE*, *simp*, *elim conjE*)

Oh man, it generates funny subgoals. Let's try generalising and simplifying pre. Where is the contradiction?

unfolding *pre-sum-elems-def*
oops

lemma *l-sg1-1-missing-assumption*:
$\textit{pre-play-move } p\ m\ s \Longrightarrow \textit{sum-elems } s \neq \textit{MAX-PILE} \Longrightarrow m + (\textit{sum-elems } s) < \textit{MAX-PILE}$
unfolding *pre-play-move-def*
unfolding *moves-left-def pre-moves-left-def*
apply (*simp*)
apply (*cases m=1*, *simp-all*)
unfolding *inv-Moves-def Let-def*
apply *simp-all*
apply (*case-tac sum-elems s* $\neq$ *19*, *simp-all*)
apply (*elim conjE*, *simp*)
defer
apply (*elim conjE impE*, *simp-all*)

different strategy on m case analysis didn't work. Generalise

oops

lemma *l-sg1-1-missing-assumption-simplified*:
$0 < m \Longrightarrow m \leq \textit{MAX-MOV} \Longrightarrow$
$(\textit{MAX-PILE} - \textit{sum-elems } s \neq 1 \longrightarrow m < \textit{MAX-PILE} - \textit{sum-elems } s) \Longrightarrow$
$(\textit{MAX-PILE} - \textit{sum-elems } s = m \longrightarrow m = 1) \Longrightarrow$
$\textit{sum-elems } s \neq \textit{MAX-PILE} \Longrightarrow m + \textit{sum-elems } s < \textit{MAX-PILE}$
apply *safe*
nitpick[*user-axioms*]

Now nicpick found a counter exmple :-(. We were too aggresive in the assumption simplification. If we play in VDM the counter example we see why (assuming in NimFull.vdmsl):

```
> p let s = [3,3,3,3,3,3,1], s'=play_move(<P2>, 1, s) in
    moves_left(s')
= 0
Executed in 0.007 secs.
> p let s = [3,3,3,3,3,3,1], s'=play_move(<P2>, 3, s) in
    moves_left(s')
Error 4060: Type invariant violated for Moves in 'NimFull' (
    console) at line 1:29
MainThread>
```

1. $[\![0 < m; m \leq \textit{MAX-MOV}; \textit{sum-elems } s \neq \textit{MAX-PILE};$
   $\neg\, m + \textit{sum-elems } s < \textit{MAX-PILE}; \textit{MAX-PILE} - \textit{sum-elems } s = 1;$
   $\textit{MAX-PILE} - \textit{sum-elems } s \neq m]\!]$
   $\Longrightarrow \textit{False}$
2. $[\![0 < 1; 1 \leq \textit{MAX-MOV}; \textit{sum-elems } s \neq \textit{MAX-PILE};$
   $\neg\, 1 + \textit{sum-elems } s < \textit{MAX-PILE}; \textit{MAX-PILE} - \textit{sum-elems } s = 1; m = 1]\!]$
   $\Longrightarrow \textit{False}$
3. $[\![0 < m; m \leq \textit{MAX-MOV}; \textit{sum-elems } s \neq \textit{MAX-PILE}; m < \textit{MAX-PILE} - \textit{sum-elems } s;$

$MAX\text{-}PILE - sum\text{-}elems\ s \neq m]\!]$
$\Longrightarrow m + sum\text{-}elems\ s < MAX\text{-}PILE$
4. $[\![0 < 1;\ 1 \leq MAX\text{-}MOV;\ sum\text{-}elems\ s \neq MAX\text{-}PILE;\ 1 < MAX\text{-}PILE - sum\text{-}elems\ s;$
$m = 1]\!]$
$\Longrightarrow 1 + sum\text{-}elems\ s < MAX\text{-}PILE$

oops

lemma *l-sg1-1-missing-assumption-strengthened*:
pre-play-move p m s $\Longrightarrow m \leq$ *moves-left s* $\Longrightarrow$ *sum-elems s* $\neq$ *MAX-PILE* $\Longrightarrow m +$ (*sum-elems s*) $<$ *MAX-PILE*
unfolding *pre-play-move-def*
unfolding *moves-left-def pre-moves-left-def*
apply (*safe*,*simp-all*)
unfolding *post-fair-play-def pre-fair-play-def pre-who-plays-next-def post-who-plays-next-def*

isFirst-def fair-play-def who-plays-next-def
apply *simp*
unfolding *inv-Moves-def inv-Move-def post-sum-elems-def pre-sum-elems-def inv-Player-def inv-VDMNat1-def inv-VDMNat-def*
apply (*safe*, *simp*)

Another term is missing when m = 1? What could it be?

1. $[\![p = (\mathit{if}\ len\ s\ mod\ 2 = 0\ \mathit{then}\ P1\ \mathit{else}\ P2);\ sum\text{-}elems\ s = 19;\ m = 1;$
$inv\text{-}SeqElems\ (\lambda m.\ 0 < m \wedge m \leq MAX\text{-}MOV)\ s;\ s \neq [\,]]\!]$
$\Longrightarrow False$

oops

AHA!!!: to play there must be more moves left? This means the final play specification isn't quite right: the precondition of *play-move* is wrong: it needs to be when = 1 not m!

Part of the difficulty in *play-move* is that its precondition has too many (assymetric) cases. Let's try with a simpler, more uniform scenario.

thm *pre-play-move-def*
definition
pre-play-move-NEW :: *Player* $\Rightarrow$ *Move* $\Rightarrow$ *Moves* $\Rightarrow \mathbb{B}$
where
pre-play-move-NEW p m s $\equiv$
inv-Player p $\wedge$
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*) $\wedge$
$0 <$ *moves-left s* $\wedge$
$m \leq$ *moves-left s* $\wedge$

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

lemma *l-sg1-1-missing-assumption-strengthened*:
pre-play-move-NEW p m s $\Longrightarrow m + (sum\text{-}elems\ s) \leq MAX\text{-}PILE$
unfolding *pre-play-move-NEW-def*
unfolding *moves-left-def pre-moves-left-def*
apply (*elim conjE*)
apply (*simp only*: *le-less*)
apply (*elim disjE*)
apply *simp-all*
done

lemma *l-sg1-1-inv-Moves-sum-elems-append*:
inv-Move m $\Longrightarrow$
pre-play-move-NEW p m s $\Longrightarrow$
inv-SeqElems inv-Move s $\Longrightarrow$
pre-sum-elems s $\Longrightarrow$
post-sum-elems s (*sum-elems s*) $\Longrightarrow$
moves-left s $\neq 1 \Longrightarrow$ *sum-elems s* $\neq$ *MAX-PILE* $\Longrightarrow$ *sum-elems* (*s* @ [*m*]) $\leq$ *MAX-PILE*
apply (*simp add*: *l-sum-elems-append-gen*)
unfolding *pre-play-move-NEW-def*
by (*simp add*: *moves-left-def*)

The last lemma ⟦*inv-Move ?m*; *pre-play-move-NEW ?p ?m ?s*; *inv-SeqElems inv-Move ?s*; *pre-sum-elems ?s*; *post-sum-elems ?s* (*sum-elems ?s*); *moves-left ?s* $\neq 1$; *sum-elems ?s* $\neq$ *MAX-PILE*⟧ $\Longrightarrow$ *sum-elems* (*?s* @ [*?m*]) $\leq$ *MAX-PILE* is also useful to prove a few others associated with the original *pre-play-move* definition.

lemma *l-sg1-2-inv-Moves-sum-elems-append*:
pre-play-move p m s $\Longrightarrow$ *moves-left s* $\geq m \Longrightarrow$ *sum-elems s* $\leq$ *MAX-PILE* $\Longrightarrow$ *s* $ *len s* = *1* $\Longrightarrow$ *sum-elems* (*s* @ [*m*]) $\leq$ *MAX-PILE*
using *l-sg1-1-inv-Moves-sum-elems-append l-sum-elems-append moves-left-def pre-play-move-def*

apply *simp*
done

For the final case, we start with the naive attempt from remaining goal as

lemma *l-sg1-inv-Moves-append*: *inv-Move m* $\Longrightarrow$ *inv-Moves s* $\Longrightarrow$ *pre-play-move p m s* $\Longrightarrow$ *inv-Moves* (*s* @ [*m*])
unfolding *inv-Moves-def Let-def*
apply (*simp*,*safe*)

This generates (10) new subgoals (some somewhat repeated), which some Sledgehammer already finds proofs for

apply (*simp add*: *l-inv-SeqElems-append*)
apply (*simp add*: *l-inv-SeqElems-append pre-sum-elems-def*)
apply (*metis l-inv-SeqElems-append l-sg4-post-sume l-sum-elems-append*)
defer

using *l-applyVDMSeq-append-last l-sum-elems-append moves-left-def pre-play-move-def*
apply *force*
apply (*simp add*: *l-inv-SeqElems-append*)
apply (*simp add*: *l-inv-SeqElems-append pre-sum-elems-def*)
apply (*simp add*: *inv-Moves-def l-sg5-post-sume-append*)
defer
using *l-applyVDMSeq-append-last l-sum-elems-append moves-left-def pre-play-move-def*
apply *force*
apply (*subgoal-tac moves-left s* $\geq$ *m*)
defer
apply (*subgoal-tac moves-left s* $\geq$ *m*)
using *l-sg1-2-inv-Moves-sum-elems-append* **apply** *blast*
unfolding *moves-left-def pre-play-move-def inv-Moves-def*
apply *simp-all*

missing condition on precondition of *play-move*

oops

With the sorry proof for case above, we know that there is something wrong with precondition of *play-move* that needs fixing. I will leave this as an exercise.

lemma *pre-play-move p m s* = *pre-play-move-NEW p m s*
apply *safe*
unfolding *pre-play-move-def pre-play-move-NEW-def*
apply *simp-all*
unfolding *moves-left-def*
apply *simp-all*
apply (*safe*, *simp-all*)
defer
using *l-inv-Move-nat1* **apply** *force*
apply (*insert l-inv-Move-nat1*[*of m*], *simp*)
oops

Older has a missing case: when at end of play but m is 2 or 3!

lemma *pre-play-move p m s* $\Longrightarrow$ *pre-play-move-NEW p m s*
unfolding *pre-play-move-def pre-play-move-NEW-def*
apply *simp-all*
unfolding *moves-left-def*
apply *simp-all*
apply (*safe*, *simp-all*)
defer
using *l-inv-Move-nat1* **apply** *fastforce*

The missing case: when *moves-left s* is 1, yet nothing is said about *m* when it could have been 2 or 3.

1. $[\![$*inv-Player p*; *inv-Move m*; *inv-Moves s*; *pre-moves-left s*;
pre-fair-play p s; *post-fair-play p s True*; *fair-play p s*;
sum-elems s = *19*; *m* $\neq$ *1*$]\!]$
$\Longrightarrow$ *m* $\leq$ *1*

unfolding *inv-Move-def inv-VDMNat1-def*
apply *safe*
apply (*cases m=1*, *simp-all*)
oops

New version is stronger than older, hence covers the case

lemma *pre-play-move-NEW p m s* $\Longrightarrow$ *pre-play-move p m s*
unfolding *pre-play-move-def pre-play-move-NEW-def*
apply *simp-all*
unfolding *moves-left-def*
apply *simp-all*
by (*safe*, *simp-all*)

lemma *inv-Moves s* $\Longrightarrow$ *sum-elems s* = *MAX-PILE* − *1* $\Longrightarrow$ *pre-play-move p 2 s*
unfolding *pre-play-move-def*
apply *simp*
unfolding *pre-fair-play-def pre-who-plays-next-def isFirst-def post-fair-play-def post-who-plays-next-def inv-Player-def*
apply *simp*
unfolding *pre-moves-left-def pre-sum-elems-def*
apply *simp*
apply *safe*
apply (*simp add*: *inv-Move-def inv-VDMNat1-def*)
apply (*simp add*: *l-inv-Moves-inv-SeqElems*)
apply (*simp add*: *moves-left-def*)+
unfolding *fair-play-def who-plays-next-def*

The statement works for *P2* not *P1*.

1. ⟦*inv-Moves s*; *sum-elems s* = *19*⟧
$\Longrightarrow$ *p* = (*if len s vdmmod 2* = *0 then P1 else P2*)

oops

lemma *inv-Moves s* $\Longrightarrow$ *sum-elems s* = *MAX-PILE* − *1* $\Longrightarrow$ *pre-play-move P2 2 s*
unfolding *pre-play-move-def*
apply *simp*
unfolding *pre-fair-play-def pre-who-plays-next-def isFirst-def post-fair-play-def post-who-plays-next-def inv-Player-def*
apply *simp*
unfolding *pre-moves-left-def pre-sum-elems-def*
apply *simp*
apply *safe*
apply (*simp add*: *inv-Move-def inv-VDMNat1-def*)
apply (*simp add*: *l-inv-Moves-inv-SeqElems*)
apply (*simp add*: *moves-left-def*)+
unfolding *fair-play-def who-plays-next-def*
apply *simp*
unfolding *inv-Moves-def post-sum-elems-def pre-sum-elems-def Let-def inv-VDMNat-def*
apply *simp*

apply *safe*

1. $[\![$*sum-elems s* $= 19$; *inv-SeqElems inv-Move s*; $s \neq []]\!] \Longrightarrow$ *len s mod 2* $= 1$

To prove this will be involved: we would have to show that any sequence we get that has *sum-elems s* $= 19$ has an odd length. Perhaps restate the goal again with an extra assumption.

oops

lemma *l-pre-play-move-OFFENDING-CASE*:
fair-play p s $\Longrightarrow$ *inv-Moves s* $\Longrightarrow$ *sum-elems s* $=$ *MAX-PILE* $- 1 \Longrightarrow$ *pre-play-move p 2 s*
unfolding *pre-play-move-def*
apply *simp*
unfolding *pre-fair-play-def pre-who-plays-next-def isFirst-def post-fair-play-def post-who-plays-next-def inv-Player-def*
pre-moves-left-def pre-sum-elems-def
apply *simp*
by (*simp add*: *inv-Move-def inv-VDMNat1-def l-inv-Moves-inv-SeqElems moves-left-def*)

lemma *l-pre-play-move-NEW-OFFENDING-CASE-SOLUTION*:
fair-play p s $\Longrightarrow$ *inv-Moves s* $\Longrightarrow$ *sum-elems s* $=$ *MAX-PILE* $- 1 \Longrightarrow$ *pre-play-move-NEW p 2 s*
unfolding *pre-play-move-NEW-def*
apply *simp*
unfolding *pre-fair-play-def pre-who-plays-next-def isFirst-def post-fair-play-def post-who-plays-next-def inv-Player-def*
pre-moves-left-def pre-sum-elems-def
apply *simp*
apply *safe*
apply (*simp add*: *inv-Move-def inv-VDMNat1-def*)
using *l-inv-Moves-inv-SeqElems* **apply** *blast*
apply (*simp add*: *moves-left-def*)
defer
using *moves-left-def* **apply** *auto*[*1*]
unfolding *moves-left-def*
apply *simp*

Finally we see that *pre-play-move-NEW* fixes the offending case.

oops

The general case missing by the original prcondition is when we are at the end of the game but the call comes with *m* different from $1 :: {}'a$.

lemma *l-pre-play-move-NEW-OFFENDING-CASE-SOLUTION-GENERAL*:
fair-play p s $\Longrightarrow$ *inv-Move m* $\Longrightarrow$ *inv-Moves s* $\Longrightarrow$ *sum-elems s* $=$ *MAX-PILE* $- 1 \Longrightarrow$ $m \neq 1 \Longrightarrow \neg$ *pre-play-move-NEW p m s*
unfolding *pre-play-move-NEW-def*
apply *simp*

unfolding *pre-fair-play-def pre-who-plays-next-def isFirst-def post-fair-play-def post-who-plays-next-def inv-Player-def*
pre-moves-left-def pre-sum-elems-def
apply *simp*
using *l-inv-Move-nat1 moves-left-def* **by** *force*

9.7 Putting it all together for satisfiability PO for *play-move*

3 Lemma-based attempt with sledgehammer support.

Other example lemmas deleted are commented in the code below. It was assuming that the lemma about append over invariant of Moves worked.

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

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

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

lemma *l-sg1-inv-Moves-end*: $(s\ @\ [m])\ !\ \textit{nat}\ (\textit{len } s) = m$
unfolding *len-def*
by *simp*

lemma $0 < \textit{moves-left } s \wedge m \leq \textit{moves-left } s \Longrightarrow \textit{inv-Move } m \Longrightarrow \textit{inv-Moves } s \Longrightarrow \textit{inv-Moves}(s\ @\ [m])$
unfolding *inv-Moves-def Let-def*
apply *simp*
apply (*intro conjI impI*)
apply (*simp add*: *l-inv-SeqElems-append*)
apply (*simp add*: *l-inv-SeqElems-append pre-sum-elems-def*)
apply (*simp add*: *inv-Moves-def l-sg5-post-sume-append*)
apply (*simp add*: *l-sum-elems-append moves-left-def*)
apply (*simp add*: *l-sg1-inv-Moves-end*)
unfolding *moves-left-def*

apply (*elim conjE*)
apply (*induct s*)
apply (*simp add*: *inv-Move-def*)
apply *simp*

Looks like a similar dead end as seen before

oops

lemma *l-sg1-1-inv-Moves-append-NEW*:
inv-Move m $\Longrightarrow$
inv-Moves s $\Longrightarrow$
0 < *moves-left s* $\Longrightarrow$ *m* $\leq$ *moves-left s* $\Longrightarrow$ *moves-left s* $\neq$ *m* $\Longrightarrow$ *inv-Moves* (*s* @ [*m*])
unfolding *moves-left-def*
apply *simp*
unfolding *inv-Moves-def Let-def*
apply *simp*
by (*metis l-inv-SeqElems-append l-sg4-post-sume l-sum-elems-append le-diff-eq less-diff-eq less-irrefl less-le pre-sum-elems-def*)

lemma *l-sg1-2-inv-Moves-append-NEW*:
inv-Move m $\Longrightarrow$
inv-Moves s $\Longrightarrow$
0 < *moves-left s* $\Longrightarrow$ *1* $\leq$ *moves-left s* $\Longrightarrow$ *inv-Moves* (*s* @ [*1*])
unfolding *inv-Moves-def Let-def*
apply *simp*

Sledgehammer finds these lemmas, which are not used/useful

thm *dbl-inc-simps*(*5*) *dbl-simps*(*3*) *dbl-simps*(*5*)

Lemma we came up with was caught here; i.e. useful to generalise for later

thm *l-concat-append*
by (*metis inv-Move-def inv-VDMNat1-def l-inv-SeqElems-append l-sg1-inv-Moves-end l-sg4-post-sume l-sum-elems-append le-diff-eq moves-left-def one-le-numeral pre-sum-elems-def zero-less-one*)

lemma *l-sg1-inv-Moves-append-NEW*: *pre-play-move-NEW p m s* $\Longrightarrow$ *inv-Moves* (*s* @ [*m*])
unfolding *pre-play-move-NEW-def*
apply *safe*
using *l-sg1-1-inv-Moves-append-NEW* **apply** *blast*
using *l-sg1-2-inv-Moves-append-NEW* **by** *blast*

theorem *PO-play-move-sat-exp-NEW-obl*
unfolding *PO-play-move-sat-exp-NEW-obl-def*
unfolding *post-play-move-NEW-def play-move-def*
apply (*simp*, *safe*)

8 instead of 12 subgoals from the first attempt

apply (*simp add*: *l-sg1-inv-Moves-append-NEW*)

apply (*simp add*: *pre-moves-left-def pre-play-move-NEW-def*)
apply (*simp add*: *l-sg1-inv-Moves-append-NEW l-sg2-pre-sume*)
apply (*meson inv-Moves-def pre-play-move-NEW-def*)
apply (*meson inv-Moves-def l-sg1-inv-Moves-append-NEW*)
apply (*simp add*: *l-inv-Move-nat1 l-sum-elems-append pre-play-move-NEW-def*)
apply (*simp add*: *l-sum-elems-append*)
by (*simp add*: *l-sg6-2-fair-play pre-play-move-NEW-def*)

Finally, the lemmas that were useful are displayed below.

thm
l-sg1-inv-Moves-append-NEW
l-sg2-pre-sume
l-inv-Move-nat1
l-sum-elems-append
l-sg6-2-fair-play

Also here, an alternative (albeit similar) proof of the same goal.

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

...

1. $\bigwedge p\ m\ s.\ \textit{pre-play-move-NEW}\ p\ m\ s \Longrightarrow \textit{inv-Moves}\ (s\ @\ [m])$
2. $\bigwedge p\ m\ s.\ \textit{pre-play-move-NEW}\ p\ m\ s \Longrightarrow \textit{pre-sum-elems}\ s$
3. $\bigwedge p\ m\ s.\ \textit{pre-play-move-NEW}\ p\ m\ s \Longrightarrow \textit{pre-sum-elems}\ (s\ @\ [m])$
4. $\bigwedge p\ m\ s.\ \textit{pre-play-move-NEW}\ p\ m\ s \Longrightarrow \textit{post-sum-elems}\ s\ (\textit{sum-elems}\ s)$
5. $\bigwedge p\ m\ s.$
   $\textit{pre-play-move-NEW}\ p\ m\ s \Longrightarrow$
   $\textit{post-sum-elems}\ (s\ @\ [m])\ (\textit{sum-elems}\ (s\ @\ [m]))$
6. $\bigwedge p\ m\ s.\ \textit{pre-play-move-NEW}\ p\ m\ s \Longrightarrow \textit{sum-elems}\ s < \textit{sum-elems}\ (s\ @\ [m])$
7. $\bigwedge p\ m\ s.\ \textit{pre-play-move-NEW}\ p\ m\ s \Longrightarrow \textit{sum-elems}\ s + m = \textit{sum-elems}\ (s\ @\ [m])$
8. $\bigwedge p\ m\ s.\ \textit{pre-play-move-NEW}\ p\ m\ s \Longrightarrow \neg\,\textit{fair-play}\ p\ (s\ @\ [m])$

apply (*simp add*: *l-sg1-inv-Moves-append-NEW*)
using *l-sg2-pre-sume pre-play-move-NEW-def* **apply** *blast*
using *l-sg3-pre-sume-append pre-play-move-NEW-def* **apply** *blast*
apply (*meson inv-Moves-def pre-play-move-NEW-def*)
apply (*simp add*: *l-sg5-post-sume-append pre-play-move-NEW-def*)
apply (*simp add*: *l-inv-Move-nat1 l-sum-elems-append pre-play-move-NEW-def*)
apply (*simp add*: *l-sum-elems-append*)
by (*simp add*: *l-sg6-2-fair-play pre-play-move-NEW-def*)

10 VDM Operations satisfiability POs

theorem *PO-first-player-winning-choose-move-sat-exp-obl0*
unfolding *PO-first-player-winning-choose-move-sat-exp-obl0-def*
apply (*intro allI impI*)
unfolding *pre-first-player-winning-choose-move0-def*
post-first-player-winning-choose-move0-def
apply (*elim conjE*)

unfolding *post-fixed-choose-move-def first-player-winning-choose-move-def*
apply *simp*
apply (*intro conjI*)
unfolding *inv-Move-def max-def Let-def*

too repetitive on the various appearances of *inv-Move*

oops
find-theorems *sum-elems* (- @ -)

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

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

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

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

doesn't work every time;

oops

You can name group of lemmas

lemmas *PO-first-player-winning-choose-move-sat-exp-obl0-pre-post* =
PO-first-player-winning-choose-move-sat-exp-obl0-def
pre-first-player-winning-choose-move0-def
post-first-player-winning-choose-move0-def
post-fixed-choose-move-def

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*: $0 < best\text{-}move\ ms \Longrightarrow inv\text{-}Move\ (best\text{-}move\ ms)$
unfolding *best-move-def moves-left-def inv-Move-def inv-VDMNat1-def*
by *simp*

theorem *PO-first-player-winning-choose-move-sat-exp-obl0*
unfolding *PO-first-player-winning-choose-move-sat-exp-obl0-pre-post first-player-winning-choose-move-def*
apply (*safe,simp*)

first goal saying that resut must be *inv-Move*, but that's only the case if *best-move* isn't zero!
Given lemma above $0 < best\text{-}move\ ?ms \Longrightarrow inv\text{-}Move\ (best\text{-}move\ ?ms)$, it's a missing PRECONDITION!

defer
apply (*simp add*: *l-best-move-range*)

similar to *inv-Move*, don't want to keep expanding *inv-Nim*

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*

find-theorems *name*:*split name*:*if*
thm *Let-def option*.*split split-ifs*

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*)

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 impI*)
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*)
apply (*elim conjE*, *intro conjI impI*)
using *inv-VDMNat-def l-inv-Move-nat1* **apply** *force*
using *l-inv-Move-nat1* **apply** *force*
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 first-player-winning-choose-move-def*
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*)

realise that *1* $\leq$ *best-move ?ms* $\Longrightarrow$ *best-move ?ms* $\leq$ *MAX-MOV* can be generalised for *inv-Move*!

oops

theorem *PO-first-player-winning-choose-move-sat-exp-obl*
unfolding *PO-first-player-winning-choose-move-sat-exp-obl-pre-post first-player-winning-choose-move-def*
  apply (*intro allI impI*, *elim conjE*, *intro conjI*, *simp-all*)
  apply (*simp add*: *l-first-player-win-best-move*)
unfolding *Let-def*
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-first-player-win-best-move l-inv-Move-nat1 f-Nim-inv-Moves*)

Another interesting lemma opportunity

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*)

We still need it but, the side conditions are not right yet

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-first-player-win-best-move*)
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*)

Wahh! Complicated. We need more lemmas for this one

oops

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

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

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*: *vdmmod-mod-ge0 l-best-move-mov-limit-mod*)

Let's try and reuse the lemmas everywhere, at once; plus expanding the easy case on *will-first-player-win* as well as Isabelle constructs for max and let. It works: makes for two sub goals.

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 Let-def*
pre-who-plays-next-def pre-moves-left-def pre-play-move0-def will-first-player-win-def first-player-winning-choose-move-def
apply (*simp-all add*: *l-first-player-win-best-move l-inv-Move-nat1 l-isFirst l-best-move-inv*)
apply (*safe*)

Argh...! seems like we are back to where we strated. Perhaps the first goal to tackle should be the hard, last one

oops

lemma *l-sg-1*: *inv-Nim bst* $\Longrightarrow$
inv-Moves (*moves bst*) $\wedge$ *pre-sum-elems* (*moves bst*) $\Longrightarrow$
$0 <$ *moves-left* (*moves bst*) $\Longrightarrow 1 <$ *moves-left* (*moves bst*)
unfolding *inv-Nim-def inv-Nim-flat-def* **apply** *simp*

Nim bst is irrelevant; abstract

oops

lemma *PO-first-player-winning-choose-move-sat-exp-obl*
unfolding *PO-first-player-winning-choose-move-sat-exp-obl-pre-post Let-def*
apply (*intro allI impI*, *elim conjE*, *intro conjI*, *simp-all*)
unfolding *max-def*
pre-who-plays-next-def pre-moves-left-def pre-play-move0-def will-first-player-win-def first-player-winning-choose-move-def
apply (*simp add*: *l-first-player-win-best-move*)

Don't know where the offending goal is coming from yet

oops

lemma *PO-first-player-winning-choose-move-sat-exp-obl*
unfolding *PO-first-player-winning-choose-move-sat-exp-obl-pre-post Let-def*
apply (*intro allI impI*, *elim conjE*, *intro conjI*, *simp-all*)

unfolding *first-player-winning-choose-move-def*
apply (*simp add*: *l-first-player-win-best-move l-inv-Move-nat1*)
unfolding *pre-who-plays-next-def best-move-def inv-Move-def*
apply (*simp add*: *l-isFirst f-Nim-inv-Moves inv-VDMNat1-def*)
apply (*simp add*: *l-moves-left-sat*)
apply (*simp add*: *l-isFirst pre-moves-left-def*)
apply (*simp add*: *inv-Player-def post-who-plays-next-def*)
using *pre-will-first-player-win-def inv-MAX-PILE-def inv-VDMNat1-def* **apply** *simp*

not via sledgehammer; try simplifying all

apply *simp-all*
unfolding *play-move-def pre-play-move-def pre-best-move-def*
apply *simp-all*

goals 2,3, 4 show we need lemmas about *inv-Moves* and *pre-moves-left* over concatenation

oops

lemma *l-inv-Moves-Append*: *inv-Moves* (*s* @ *t*) = (*inv-Moves s* ∧ *inv-Moves t*)
apply (*induct s*)
apply *simp*

two cases as empty and non-empty

oops

lemma *l-inv-Moves-Empty*: *inv-Moves* []
unfolding *inv-Moves-defs pre-sum-elems-def elems-def inv-VDMNat-def*
by *simp*

lemma *l-inv-Moves-Append*: *inv-Moves* (*s* @ *t*) = (*inv-Moves s* ∧ *inv-Moves t*)
apply (*induct s*)
apply (*simp add*: *l-inv-Moves-Empty*)

for append, need lemma about cons

oops

lemma *l-inv-Moves-Cons*: *inv-Moves* (*a#s*) = (*inv-Move a* ∧ *inv-Moves s*)
apply (*rule iffI*)
unfolding *inv-Moves-def*
apply *auto*

needs to go slowly

oops

lemma *l-inv-Moves-Cons*: *inv-Moves* (*a#s*) = (*inv-Move a* ∧ *inv-Moves s*)
apply (*safe*)
apply (*simp add*: *inv-Moves-defs*(*1*) *l-inv-SeqElems-Cons*)
unfolding *inv-Moves-def pre-sum-elems-def Let-def*
apply (*safe*)
apply (*simp-all add*: *l-inv-SeqElems-Cons*)

unfolding *post-sum-elems-def*
apply *simp-all*

needs slower pace

oops

Singleton version of *inv-Moves* equal *inv-Move*

lemma *l-inv-Moves-Singleton*: *inv-Moves* [*m*] = *inv-Move m*
unfolding *inv-Moves-def inv-SeqElems-def*
apply *simp*
unfolding *pre-sum-elems-def Let-def post-sum-elems-def*
apply *simp*
using *inv-Move-def inv-VDMNat-def l-inv-Move-nat1 l-inv-SeqElems-Cons* **by** *fastforce*

Singleton version of *inv-Moves* append. See also *pre-play-move-NEW ?p ?m ?s* $\Longrightarrow$ *inv-Moves* (*?s* @ [*?m*])

lemma *l-inv-Moves-Append1*: *inv-Moves* (*s* @ [*m*]) = (*inv-Moves s* $\wedge$ *inv-Move m*)
find-theorems - @ [-] *name*:*List*
apply (*induct s*)
apply (*simp add*: *l-inv-Moves-Empty l-inv-Moves-Singleton*)
unfolding *inv-Moves-def*
apply (*simp*)
oops

lemma *l-not-inv-Move-zero*: $\neg$ *inv-Move 0*
by (*simp add*: *inv-Move-def inv-VDMNat1-def*)

lemma *l-inv-Moves-Cons*: *inv-Moves* (*a#s*) $\Longrightarrow$ (*inv-Move a* $\wedge$ *inv-Moves s*)
unfolding *inv-Moves-def post-sum-elems-def pre-sum-elems-def Let-def*
apply (*simp add*: *l-inv-SeqElems-Cons*)
using *inv-VDMNat-def l-inv-Move-nat1 l-pre-sum-elems* **by** *fastforce*

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

safe will be unhelpful here as it will generate manye (13) small goals

apply *safe*
apply (*simp add*: *l-first-player-win-best-move pre-first-player-winning-choose-move-def*)
unfolding *pre-who-plays-next-def*
apply (*simp add*: *pre-first-player-winning-choose-move-def*)+
unfolding *pre-moves-left-def*
apply (*simp add*: *l-isFirst*)
apply (*simp add*: *l-moves-left-sat pre-moves-left-def*)
apply (*simp add*: *f-Nim-inv-Moves isFirst-def pre-first-player-winning-choose-move-def*)
apply (*simp add*: *inv-Player-def post-who-plays-next-def*)

Sledgehammer seems to have stopped being useful. Expansion of various parts for this goal is unhelpful. make it the first goal

oops

lemma *l-sg-ml*: $\mathit{inv\text{-}Nim}\ \mathit{bst} \Longrightarrow$
$\mathit{inv\text{-}Moves}\ (\mathit{moves}\ \mathit{bst}) \wedge \mathit{pre\text{-}sum\text{-}elems}\ (\mathit{moves}\ \mathit{bst}) \Longrightarrow$
$0 < \mathit{moves\text{-}left}\ (\mathit{moves}\ \mathit{bst}) \Longrightarrow \mathit{Suc}\ 0 < \mathit{moves\text{-}left}\ (\mathit{moves}\ \mathit{bst})$
oops

lemma *l-sg-let*: $\mathit{inv\text{-}Nim}\ \mathit{bst} \Longrightarrow$
$\mathit{pre\text{-}moves\text{-}left}\ (\mathit{moves}\ \mathit{bst}) \Longrightarrow$
$0 < \mathit{moves\text{-}left}\ (\mathit{moves}\ \mathit{bst}) \Longrightarrow$
$\mathit{let}\ \mathit{pm} = \mathit{play\text{-}move}\ (\mathit{current}\ \mathit{bst})\ (\mathit{max}\ (\mathit{Suc}\ 0)\ (\mathit{best\text{-}move}\ (\mathit{moves}\ \mathit{bst})))\ (\mathit{moves}$
$\mathit{bst})$
$\mathit{in}\ \mathit{pre\text{-}best\text{-}move}\ \mathit{pm} \wedge$
$\mathit{post\text{-}best\text{-}move}\ \mathit{pm}\ (\mathit{best\text{-}move}\ \mathit{pm}) \wedge (\mathit{isFirst}\ (\mathit{who\text{-}plays\text{-}next}\ (\mathit{moves}\ \mathit{bst})) \longrightarrow$
$\mathit{best\text{-}move}\ \mathit{pm} = 0)$
find-theorems *name*:*let* −*name*:*Complete-* −*name*:*Induc* −*name*:*Set* −*name*:*List* −*name*:*Lat*
−*name*:*Nim* −*name*:*Map* −*name*:*BNF*
−*name*:*Predicate*
find-theorems *name*:*let* *name*:*cong*
unfolding *Let-def*
apply (*intro conjI impI*)
unfolding *pre-best-move-def*
apply (*intro conjI*)
unfolding *play-move-def*
oops

lemma *l-inv-Moves-Append1*: *inv-Moves* (*s* @ [*m*])
unfolding *inv-Moves-def*
apply (*intro conjI*)
find-theorems *inv-SeqElems* - -
oops

lemma *f-inv-Move* : $\mathit{inv\text{-}Move}\ m \Longrightarrow m \leq \mathit{MAX\text{-}MOV}$
unfolding *inv-Move-def* **by** *simp*

lemma *l-MAX-rel*: $\mathit{MAX\text{-}MOV} < \mathit{MAX\text{-}PILE}$
by *simp*

end