Formal

Cornelius Diekmann

September 20, 2022

Contents

1 Disclaimer

Ich habe

- kein Ahnung von Philosophie.

- keine Ahnung von Recht und Jura.
- und schon gar keine Ahnung von Strafrecht oder Steuerrecht.

Und in dieser Session werden ich all das zusammenwerfen.

Cheers!

2 Gesetz

Definiert einen Datentyp um Gesetzestext zu modellieren.

datatype *'a tatbestand = Tatbestand ‹'a›*

datatype *'a rechtsfolge = Rechtsfolge ‹'a›*

datatype *('a, 'b) rechtsnorm = Rechtsnorm ‹'a tatbestand› ‹'b rechtsfolge›*

datatype *'p prg = Paragraph ‹'p›*

datatype *('p, 'a, 'b) gesetz = Gesetz ‹('p prg × ('a, 'b) rechtsnorm) set›*

Beispiel, von https://de.wikipedia.org/wiki/Rechtsfolge:

```
value ‹Gesetz {
  (Paragraph ''823 BGB'',
   Rechtsnorm
     (Tatbestand ''Wer vorsaetzlich oder fahrlaessig das Leben, den Koerper, die Gesundheit, (...),
                   das Eigentum oder (...) eines anderen widerrechtlich verletzt,'')
     (Rechtsfolge ''ist dem anderen zum Ersatz des daraus entstehenden Schadens verpflichtet.'')
  ),
  (Paragraph ''985 BGB'',
   Rechtsnorm
     (Tatbestand ''Der Eigentuemer einer Sache kann von dem Besitzer'')
     (Rechtsfolge ''die Herausgabe der Sache verlangen'')
  ),
  (Paragraph ''303 StGB'',
   Rechtsnorm
     (Tatbestand ''Wer rechtswidrig eine fremde Sache beschaedigt oder zerstoert,'')
     (Rechtsfolge ''wird mit Freiheitsstrafe bis zu zwei Jahren oder mit Geldstrafe bestraft.'')
  )
  }›
```

fun *neuer-paragraph :: ‹(nat, 'a, 'b) gesetz ⇒ nat prg›* **where**
‹neuer-paragraph (Gesetz G) = Paragraph ((max-paragraph (fst ‘ G)) + 1)›

Fügt eine Rechtsnorm als neuen Paragraphen hinzu:

fun *hinzufuegen :: ‹('a,'b) rechtsnorm ⇒ (nat,'a,'b) gesetz ⇒ (nat,'a,'b) gesetz›* **where**

‹*hinzufuegen rn* (*Gesetz G*) =
(*if rn* ∈ (*snd* ‘ *G*) *then Gesetz G else Gesetz* (*insert* (*neuer-paragraph* (*Gesetz G*), *rn*) *G*))›

Moelliert ob eine Handlung ausgeführt werden muss, darf, kann, nicht muss:

datatype *sollensanordnung* = *Gebot* | *Verbot* | *Erlaubnis* | *Freistellung*

Beispiel:

lemma ‹*hinzufuegen*
(*Rechtsnorm* (*Tatbestand* ''*tb2*'') (*Rechtsfolge Verbot*))
(*Gesetz* {(*Paragraph 1*, (*Rechtsnorm* (*Tatbestand* ''*tb1*'') (*Rechtsfolge Erlaubnis*)))}) =
Gesetz
{(*Paragraph 2*, *Rechtsnorm* (*Tatbestand* ''*tb2*'') (*Rechtsfolge Verbot*)),
(*Paragraph 1*, *Rechtsnorm* (*Tatbestand* ''*tb1*'') (*Rechtsfolge Erlaubnis*))}›

3 Handlung

Beschreibt Handlungen als Änderung der Welt. Unabhängig von der handelnden Person. Wir beschreiben nur vergangene bzw. mögliche Handlungen und deren Auswirkung.

Eine Handlung ist reduziert auf deren Auswirkung. Intention oder Wollen ist nicht modelliert, da wir irgendwie die geistige Welt mit der physischen Welt verbinden müssen und wir daher nur messbare Tatsachen betrachten können.

Handlungen können Leute betreffen. Handlungen können aus Sicht Anderer wahrgenommen werden. Ich brauche nur Welt vorher und Welt nachher. So kann ich handelnde Person und beobachtende Person trennen.

datatype *'world handlung* = *Handlung* (*vorher*: ‹*'world*›) (*nachher*: ‹*'world*›)

Handlung als Funktion gewrapped. Diese abstrakte Art eine Handlung zu modelliert so ein bisschen die Absicht oder Intention.

datatype (*'person*, *'world*) *handlungF* = *HandlungF* ‹*'person* ⇒ *'world* ⇒ *'world*›

Von Außen können wir Funktionen nur extensional betrachten, d.h. Eingabe und Ausgabe anschauen. Die Absicht die sich in einer Funktion verstecken kann ist schwer zu erkennen. Dies deckt sich ganz gut damit, dass Isabelle standardmäßig Funktionen nicht printet. Eine (*'person*, *'world*) *handlungF* kann nicht geprinted werden!

fun *handeln* :: ‹*'person* ⇒ *'world* ⇒ (*'person*, *'world*) *handlungF* ⇒ *'world handlung*› **where**
‹*handeln handelnde-person welt* (*HandlungF h*) = *Handlung welt* (*h handelnde-person welt*)›

Beispiel, für eine Welt die nur aus einer Zahl bestehtÖ Wenn die Zahl kleiner als 9000 ist erhöhe ich sie, ansonsten bleibt sie unverändert.

definition ‹*beispiel-handlungf* ≡ *HandlungF* (λ*p n*. *if n* < *9000 then n*+*1 else n*)›

Da Funktionen nicht geprintet werden können, sieht *beispiel-handlungf* so aus: *HandlungF* -

4 Beispiel Person

Eine Beispielbevölkerung.

datatype *person* = *Alice* | *Bob* | *Carol* | *Eve*

Unsere Bevölkerung ist sehr endlich:

lemma *UNIV-person*: ‹*UNIV* = {*Alice*, *Bob*, *Carol*, *Eve*}›

Wir werden unterscheiden:

- *'person*: generischer Typ, erlaub es jedes Modell einer Person und Bevölkerung zu haben.
- *person*: Unser minimaler Beispielstyp, bestehend aus *Alice*, *Bob*, ...

5 Kant's Kategorischer Imperativ

Immanuel Kant

„Handle nur nach derjenigen Maxime, durch die du zugleich wollen kannst, dass sie ein allgemeines Gesetz werde."

https://de.wikipedia.org/wiki/Kategorischer_Imperativ

Meine persönliche, etwas utilitaristische, Interpretation.

5.1 Maxime

Modell einer *Maxime*: Eine Maxime in diesem Modell beschreibt ob eine Handlung in einer gegebenen Welt gut ist.
Faktisch ist eine Maxime

- *'person*: die handelnde Person, i.e., *ich*.

- *'world handlung*: die zu betrachtende Handlung.
- *bool*: Das Ergebnis der Betrachtung. *True* = Gut; *False* = Schlecht.

Wir brauchen sowohl die *'world handlung* als auch die handelnde *'person*, da es einen großen Unterschied machen kann ob ich selber handel, ob ich Betroffener einer fremden Handlung bin, oder nur Außenstehender.

datatype (*'person*, *'world*) *maxime* = *Maxime* ‹*'person* ⇒ *'world handlung* ⇒ *bool*›

Beispiel

definition *maxime-mir-ist-alles-recht* :: ‹(*'person*, *'world*) *maxime*› **where**
‹*maxime-mir-ist-alles-recht* ≡ *Maxime* (λ- -. *True*)›

Um eine Handlung gegen eine Maxime zu testen fragen wir uns:

- Was wenn jeder so handeln würde?
- Was wenn jeder diese Maxime hätte? Bsp: stehlen und bestohlen werden.

definition *bevoelkerung* :: ‹*'person set*› **where** ‹*bevoelkerung* ≡ *UNIV*›
definition *wenn-jeder-so-handelt*
:: ‹*'world* ⇒ (*'person*, *'world*) *handlungF* ⇒ (*'world handlung*) *set*›
where
‹*wenn-jeder-so-handelt welt handlung* ≡
(λ*handelnde-person*. *handeln handelnde-person welt handlung*) ‘ *bevoelkerung*›
fun *was-wenn-jeder-so-handelt-aus-sicht-von*
:: ‹*'world* ⇒ (*'person*, *'world*) *handlungF* ⇒ (*'person*, *'world*) *maxime* ⇒ *'person* ⇒ *bool*›
where
‹*was-wenn-jeder-so-handelt-aus-sicht-von welt handlung* (*Maxime m*) *betroffene-person* =
(∀ *h* ∈ *wenn-jeder-so-handelt welt handlung*. *m betroffene-person h*)›

definition *teste-maxime* ::
‹*'world* ⇒ (*'person*, *'world*) *handlungF* ⇒ (*'person*, *'world*) *maxime* ⇒ *bool*› **where**
‹*teste-maxime welt handlung maxime* ≡
∀ *p* ∈ *bevoelkerung*. *was-wenn-jeder-so-handelt-aus-sicht-von welt handlung maxime p*›

Faktisch bedeutet diese Definition, wir bilden das Kreuzprodukt Bevölkerung x Bevölkerung, wobei jeder einmal als handelnde Person auftritt und einmal als betroffene Person.

lemma *teste-maxime-unfold*:
‹*teste-maxime welt handlung* (*Maxime m*) =
(∀ *p*∈*bevoelkerung*. ∀ *x*∈*bevoelkerung*. *m p* (*handeln x welt handlung*))›
lemma ‹*teste-maxime welt handlung* (*Maxime m*) =
(∀ (*p1*,*p2*)∈*bevoelkerung*×*bevoelkerung*. *m p1* (*handeln p2 welt handlung*))›

Hier schlägt das Programmiererherz höher: Wenn *'person* aufzählbar ist haben wir ausführbaren Code: *teste-maxime* = *teste-maxime-exhaust enum-class.enum* wobei *teste-maxime-exhaust* implementiert ist als *teste-maxime-exhaust bevoelk welt handlung maxime* ≡ *case maxime of Maxime m* ⇒ *list-all* (λ(*p*, *x*). *m p* (*handeln x welt handlung*)) (*List.product bevoelk bevoelk*).

5.1.1 Beispiel

Beispiel: Die Welt sei nur eine Zahl und die zu betrachtende Handlung sei, dass wir diese Zahl erhöhen. Die Mir-ist-alles-Recht Maxime ist hier erfüllt:

lemma ‹*teste-maxime*
(*42*::*nat*)
(*HandlungF* (λ(*person*::*person*) *welt*. *welt* + *1*))
maxime-mir-ist-alles-recht›

Beispiel: Die Welt ist modelliert als eine Abbildung von Person auf Besitz. Die Maxime sagt, dass Leute immer mehr oder gleich viel wollen, aber nie etwas verlieren wollen. In einer Welt in der keiner etwas hat, erfuellt die Handlung jemanden 3 zu geben die Maxime.

lemma ‹*teste-maxime*
[*Alice* ↦ (*0*::*nat*), *Bob* ↦ *0*, *Carol* ↦ *0*, *Eve* ↦ *0*]
(*HandlungF* (λ*person welt*. *welt*(*person* ↦ *3*)))
(*Maxime* (λ*person handlung*.
(*the* ((*vorher handlung*) *person*)) ≤ (*the* ((*nachher handlung*) *person*))))›

Wenn nun *Bob* allerdings bereits 4 hat, würde die obige Handlung ein Verlust für ihn bedeuten und die Maxime ist nicht erfüllt.

lemma ‹¬ *teste-maxime*
[*Alice* ↦ (*0*::*nat*), *Bob* ↦ *4*, *Carol* ↦ *0*, *Eve* ↦ *0*]
(*HandlungF* (λ*person welt*. *welt*(*person* ↦ *3*)))
(*Maxime* (λ*person handlung*.
(*the* ((*vorher handlung*) *person*)) ≤ (*the* ((*nachher handlung*) *person*))))›

5.2 Allgemeines Gesetz Ableiten

Versuch ein allgemeines Gesetz abzuleiten: TODO: Nur aus einer von außen betrachteten Handlung und einer Entscheidung ob diese Handlung ausgeführt werden soll wird es schwer ein allgemeines Gesetz abzuleiten.

type-synonym ('*world*, '*a*, '*b*) *allgemeines-gesetz-ableiten* =
‹'*world handlung* ⇒ *sollensanordnung* ⇒ ('*a*, '*b*) *rechtsnorm*›

5.3 Implementierung Kategorischer Imperativ.

Handle nur nach derjenigen Maxime, durch die du zugleich wollen kannst, dass sie ein allgemeines Gesetz werde.

Parameter

- '*person*: handelnde Person
- '*world*: Die Welt in ihrem aktuellen Zustand

- (*'person*, *'world*) *handlungF*: Eine mögliche Handlung, über die wir entscheiden wollen ob wir sie ausführen sollten.
- (*'person*, *'world*) *maxime*: Persönliche Ethik?
- (*'world*, *'a*, *'b*) *allgemeines-gesetz-ableiten*: wenn man keinen Plan hat wie man sowas implementiert, einfach als Eingabe annehmen.
- (*nat*, *'a*, *'b*) *gesetz*: Allgemeines Gesetz (für alle Menschen) Ergebnis: *sollensanordnung*: Sollen wir die Handlung ausführen? (*nat*, *'a*, *'b*) *gesetz*: Soll das allgemeine Gesetz entsprechend angepasst werden?

definition *kategorischer-imperativ* ::
‹*'person* ⇒ *'world* ⇒ (*'person*, *'world*) *handlungF* ⇒
(*'person*, *'world*) *maxime* ⇒ (*'world*, *'a*, *'b*) *allgemeines-gesetz-ableiten* ⇒
(*nat*, *'a*, *'b*) *gesetz*
⇒ (*sollensanordnung* × (*nat*, *'a*, *'b*) *gesetz*)› **where**

‹*kategorischer-imperativ ich welt handlung maxime gesetz-ableiten gesetz* ≡
let soll-handeln = if teste-maxime welt handlung maxime
then
Erlaubnis
else
Verbot in
(
soll-handeln,
hinzufuegen (*gesetz-ableiten* (*handeln ich welt handlung*) *soll-handeln*) *gesetz*
)
›

6 Simulation

datatype (*'person*, *'world*, *'a*, *'b*) *simulation-constants* = *SimConsts*
'person — handelnde Person

(*'person*, *'world*) *maxime*
(*'world*, *'a*, *'b*) *allgemeines-gesetz-ableiten*

simulate one (*'person*, *'world*) *handlungF* once

fun *simulate-handlungF*
:: (*'person*, *'world*, *'a*, *'b*) *simulation-constants* ⇒
(*'person*, *'world*) *handlungF* ⇒ *'world* ⇒ (*nat*, *'a*, *'b*) *gesetz*
⇒ (*'world* × (*nat*, *'a*, *'b*) *gesetz*)
where
simulate-handlungF (*SimConsts person maxime aga*) *h welt g* =
(*let* (*sollensanordnung*, *g'*) = *kategorischer-imperativ person welt h maxime aga g in*

```
    let w' = (if sollensanordnung = Erlaubnis
              then
                nachher (handeln person welt h)
              else
                welt
             ) in
    (w', g')
  )
```

lemma ‹*simulate-handlungF*
```
    (SimConsts
      ()
      (Maxime (λ- -. True))
      (λh s. Rechtsnorm (Tatbestand h) (Rechtsfolge ''count'')))
    (HandlungF (λp w. w+1))
    (32::int)
    (Gesetz {})=
 (33,
  Gesetz
   {(Paragraph (Suc 0), Rechtsnorm (Tatbestand (Handlung 32 33)) (Rechtsfolge ''count''))})›
```

Funktion begrenzt oft anwenden bis sich die Welt nicht mehr ändert. Parameter

- Funktion
- Maximale Anzahl Iterationen (Simulationen)
- Initialwelt
- Initialgesetz

fun *converge*
```
  :: ('world ⇒ 'gesetz ⇒ ('world × 'gesetz)) ⇒ nat ⇒ 'world ⇒ 'gesetz ⇒ ('world × 'gesetz)
```
where
```
    converge - 0       w g = (w, g)
  | converge f (Suc its) w g =
      (let (w', g') = f w g in
        if w = w' then
          (w, g')
        else
          converge f its w' g')
```

Example: Count 32..42, where *32* is the initial world and we do *10* iterations.

lemma ‹*converge* (λw g. (w+1, w#g)) *10* (*32*::*int*) ([]) =
```
      (42, [41, 40, 39, 38, 37, 36, 35, 34, 33, 32])›
```

simulate one ('*person*, '*world*) *handlungF* a few times

definition *simulateOne*

```
  :: ('person, 'world, 'a, 'b) simulation-constants ⇒
     nat ⇒ ('person, 'world) handlungF ⇒ 'world ⇒ (nat, 'a, 'b) gesetz
     ⇒ (nat, 'a, 'b) gesetz
  where
  simulateOne simconsts i h w g ≡
    let (welt, gesetz) = converge (simulate-handlungF simconsts h) i w g in
        gesetz
```

Example: Count 32..42

```
lemma ‹simulateOne
      (SimConsts () (Maxime (λ- -. True)) (λh s. Rechtsnorm (Tatbestand h) (Rechtsfolge ''count'')))
      10 (HandlungF (λp n. Suc n))
      32
      (Gesetz {}) =
  Gesetz
  {(Paragraph 10, Rechtsnorm (Tatbestand (Handlung 41 42)) (Rechtsfolge ''count'')),
   (Paragraph 9, Rechtsnorm (Tatbestand (Handlung 40 41)) (Rechtsfolge ''count'')),
   (Paragraph 8, Rechtsnorm (Tatbestand (Handlung 39 40)) (Rechtsfolge ''count'')),
   (Paragraph 7, Rechtsnorm (Tatbestand (Handlung 38 39)) (Rechtsfolge ''count'')),
   (Paragraph 6, Rechtsnorm (Tatbestand (Handlung 37 38)) (Rechtsfolge ''count'')),
   (Paragraph 5, Rechtsnorm (Tatbestand (Handlung 36 37)) (Rechtsfolge ''count'')),
   (Paragraph 4, Rechtsnorm (Tatbestand (Handlung 35 36)) (Rechtsfolge ''count'')),
   (Paragraph 3, Rechtsnorm (Tatbestand (Handlung 34 35)) (Rechtsfolge ''count'')),
   (Paragraph 2, Rechtsnorm (Tatbestand (Handlung 33 34)) (Rechtsfolge ''count'')),
   (Paragraph 1, Rechtsnorm (Tatbestand (Handlung 32 33)) (Rechtsfolge ''count''))}›
```

7 Aenderungen in Welten

```
datatype ('person, 'etwas) aenderung = Verliert 'person 'etwas | Gewinnt 'person 'etwas

definition delta-num
  :: 'person ⇒ 'etwas::{ord,minus} ⇒ 'etwas ⇒ (('person, 'etwas) aenderung) option
  where
    delta-num p i1 i2 = (
          if i1 > i2 then Some (Verliert p (i1 − i2))
      else if i1 < i2 then Some (Gewinnt p (i2 − i1))
      else None
  )

lemma delta-num p i1 i2 = Some (Gewinnt p (i::int)) ⟹ i > 0
lemma delta-num p i1 i2 = Some (Verliert p (i::int)) ⟹ i > 0
lemma delta-num p1 i1 i2 = Some (Gewinnt p2 (i::int)) ⟹ p1 = p2
lemma delta-num p1 i1 i2 = Some (Verliert p2 (i::int)) ⟹ p1 = p2
```

Deltas, d.h. Unterschiede Zwischen Welten.

Man könnte eine class Delta world einführen, mit einer delta-Funtion :: welt -> welt -> [Aenderung person

etwas] Diese Klasse würde dann Welten mit Personen und Etwas in Relation setzen. Dafür bräuchte es MultiParamTypeClasses. Eine simple Funktion ist da einfacher.

type-synonym (*'world*, *'person*, *'etwas*) *delta* =
  'world handlung ⇒ ((*'person*, *'etwas*) *aenderung*) *list*

fun *delta-num-map*
  :: ((*'person*::*enum* ⇀ (*'etwas*::{*zero*,*minus*,*ord*})), *'person*, *'etwas*) *delta*
  where
  delta-num-map (*Handlung vor nach*) =
    List.map-filter
      (λ*p*. *case* (*the-default* (*vor p*) *0*, *the-default* (*nach p*) *0*)
        of (*a*,*b*) ⇒ *delta-num p a b*)
      (*Enum.enum*::*'person list*)

lemma‹*delta-num-map*
  (*Handlung* [*Alice* ↦ *5*::*int*, *Bob* ↦ *10*, *Eve* ↦ *1*]
    [*Alice* ↦ *3*, *Bob* ↦ *13*, *Carol* ↦ *2*])
  = [*Verliert Alice 2*, *Gewinnt Bob 3*, *Gewinnt Carol 2*, *Verliert Eve 1*]›

fun *delta-num-fun*
  :: ((*'person*::*enum* ⇒ (*'etwas*::{*minus*,*ord*})), *'person*, *'etwas*) *delta*
  where
  delta-num-fun (*Handlung vor nach*) =
    List.map-filter (λ*p*. *delta-num p* (*vor p*) (*nach p*)) *Enum.enum*

lemma ‹*delta-num-fun*
  (*Handlung*
    ((λ*p*. *0*::*int*)(*Alice*:=*8*, *Bob*:=*12*, *Eve*:=*7*))
    ((λ*p*. *0*::*int*)(*Alice*:=*3*, *Bob*:=*15*, *Eve*:=*0*)))
  = [*Verliert Alice 5*, *Gewinnt Bob 3*, *Verliert Eve 7*]›

lemma *delta-num-map*: *delta-num-map* (*Handlung m1 m2*) =
    delta-num-fun (*Handlung* (λ*p*. *the-default* (*m1 p*) *0*) (λ*p*. *the-default* (*m2 p*) *0*))

8 Gesetze

8.1 Case Law

Gesetz beschreibt: (wenn vorher, wenn nachher) dann Erlaubt/Verboten, wobei vorher/nachher die Welt beschreiben. Paragraphen sind einfache *nat*

type-synonym *'world case-law* = (*nat*, (*'world* × *'world*), *sollensanordnung*) *gesetz*

Überträgt einen Tatbestand wörtlich ins Gesetz. Nicht sehr allgemein.

definition *case-law-ableiten-absolut*

```
  :: ('world, ('world × 'world), sollensanordnung) allgemeines-gesetz-ableiten
where
  case-law-ableiten-absolut handlung sollensanordnung =
    Rechtsnorm
      (Tatbestand (vorher handlung, nachher handlung))
      (Rechtsfolge sollensanordnung)

definition printable-case-law-ableiten-absolut
  :: ('world ⇒'printable-world) ⇒
    ('world, ('printable-world × 'printable-world), sollensanordnung) allgemeines-gesetz-ableiten
where
  printable-case-law-ableiten-absolut print-world h ≡
    case-law-ableiten-absolut (map-handlung print-world h)
```

Case Law etwas besser, wir zeigen nur die Änderung.

```
fun case-law-ableiten-relativ
  :: ('world handlung ⇒ (('person, 'etwas) aenderung) list)
    ⇒ ('world, (('person, 'etwas) aenderung) list, sollensanordnung)
      allgemeines-gesetz-ableiten
where
  case-law-ableiten-relativ delta handlung erlaubt =
    Rechtsnorm (Tatbestand (delta handlung)) (Rechtsfolge erlaubt)
```

9 Beispiel: Zahlenwelt

Wenn die Welt sich durch eine Zahl darstellen lässt, ...

```
datatype zahlenwelt = Zahlenwelt
    nat — verbleibend: Ressourcen sind endlich. Verbleibende Ressourcen in der Welt.
    person ⇒ int option — besitz: Besitz jeder Person.

fun gesamtbesitz :: zahlenwelt ⇒ int where
  gesamtbesitz (Zahlenwelt - besitz) = sum-list (List.map-filter besitz [Alice, Bob, Carol, Eve])

lemma gesamtbesitz (Zahlenwelt 42 [Alice ↦ 4, Carol ↦ 8]) = 12
lemma gesamtbesitz (Zahlenwelt 42 [Alice ↦ 4, Carol ↦ 4]) = 8

fun abbauen :: nat ⇒ person ⇒ zahlenwelt ⇒ zahlenwelt where
  abbauen i p (Zahlenwelt verbleibend besitz) =
    Zahlenwelt
      (verbleibend − i)
      (case besitz p
        of None ⇒ besitz(p ↦ int i)
         | Some b ⇒ besitz(p ↦ b + int i))
```

Mehr ist mehr gut. Globaler Fortschritt erlaubt stehlen, solange dabei nichts vernichtet wird.

Größer (>) anstelle (>=) ist hier echt spannend! Es sagt, dass wir nicht handeln duerfen, wenn andere nicht die Möglichkeit haben!! Das >= ist kein strenger Fortschritt, eher kein Rückschritt.

fun *globaler-fortschritt* :: *zahlenwelt handlung* ⇒ *bool* **where**
globaler-fortschritt (*Handlung vor nach*) ⟷ (*gesamtbesitz nach*) ≥ (*gesamtbesitz vor*)

Dieser globale Fortschritt sollte eigentlich allgemeines Gesetz werden und die Maxime sollte individuelle Bereicherung sein (und die unsichtbare Hand macht den Rest. YOLO).

fun *meins* :: *person* ⇒ *zahlenwelt* ⇒ *int* **where**
meins p (*Zahlenwelt verbleibend besitz*) = *the-default* (*besitz p*) *0*

fun *individueller-fortschritt* :: *person* ⇒ *zahlenwelt handlung* ⇒ *bool* **where**
individueller-fortschritt p (*Handlung vor nach*) ⟷ (*meins p nach*) ≥ (*meins p vor*)

definition *maxime-zahlenfortschritt* :: (*person*, *zahlenwelt*) *maxime* **where**
maxime-zahlenfortschritt ≡ *Maxime* (λ*ich*. *individueller-fortschritt ich*)

fun *delta-zahlenwelt* :: (*zahlenwelt*, *person*, *int*) *delta* **where**
delta-zahlenwelt (*Handlung* (*Zahlenwelt* - *vor-besitz*) (*Zahlenwelt* - *nach-besitz*)) =
Aenderung.delta-num-map (*Handlung vor-besitz nach-besitz*)

definition *sc* ≡ *SimConsts*
Alice
maxime-zahlenfortschritt
(*printable-case-law-ableiten-absolut*
(λ*w*. *case w of Zahlenwelt verbleibend besitz* ⇒ (*verbleibend*, *show-map besitz*)))
definition *sc′* ≡ *SimConsts*
Alice
maxime-zahlenfortschritt
(*case-law-ableiten-relativ delta-zahlenwelt*)

definition *initialwelt* ≡ *Zahlenwelt 42* [*Alice* ↦ *5*, *Bob* ↦ *10*]

definition *beispiel-case-law h* ≡ *simulateOne sc 20 h initialwelt* (*Gesetz* {})
definition *beispiel-case-law′ h* ≡ *simulateOne sc′ 20 h initialwelt* (*Gesetz* {})

lemma ‹*beispiel-case-law* (*HandlungF* (*abbauen 5*)) =
Gesetz
{(*Paragraph 20*,
Rechtsnorm (*Tatbestand* ((*0*, [(*Alice*, *100*), (*Bob*, *10*)]), *0*, [(*Alice*, *105*), (*Bob*, *10*)]))
(*Rechtsfolge Erlaubnis*)),
(*Paragraph 19*,
Rechtsnorm (*Tatbestand* ((*0*, [(*Alice*, *95*), (*Bob*, *10*)]), *0*, [(*Alice*, *100*), (*Bob*, *10*)]))
(*Rechtsfolge Erlaubnis*)),
(*Paragraph 18*,
Rechtsnorm (*Tatbestand* ((*0*, [(*Alice*, *90*), (*Bob*, *10*)]), *0*, [(*Alice*, *95*), (*Bob*, *10*)]))
(*Rechtsfolge Erlaubnis*)),
(*Paragraph 17*,
Rechtsnorm (*Tatbestand* ((*0*, [(*Alice*, *85*), (*Bob*, *10*)]), *0*, [(*Alice*, *90*), (*Bob*, *10*)]))

$(Rechtsfolge\ Erlaubnis)),$
$(Paragraph\ 16,$
$Rechtsnorm\ (Tatbestand\ ((0,\ [(Alice,\ 80),\ (Bob,\ 10)]),\ 0,\ [(Alice,\ 85),\ (Bob,\ 10)]))$
$(Rechtsfolge\ Erlaubnis)),$
$(Paragraph\ 15,$
$Rechtsnorm\ (Tatbestand\ ((0,\ [(Alice,\ 75),\ (Bob,\ 10)]),\ 0,\ [(Alice,\ 80),\ (Bob,\ 10)]))$
$(Rechtsfolge\ Erlaubnis)),$
$(Paragraph\ 14,$
$Rechtsnorm\ (Tatbestand\ ((0,\ [(Alice,\ 70),\ (Bob,\ 10)]),\ 0,\ [(Alice,\ 75),\ (Bob,\ 10)]))$
$(Rechtsfolge\ Erlaubnis)),$
$(Paragraph\ 13,$
$Rechtsnorm\ (Tatbestand\ ((0,\ [(Alice,\ 65),\ (Bob,\ 10)]),\ 0,\ [(Alice,\ 70),\ (Bob,\ 10)]))$
$(Rechtsfolge\ Erlaubnis)),$
$(Paragraph\ 12,$
$Rechtsnorm\ (Tatbestand\ ((0,\ [(Alice,\ 60),\ (Bob,\ 10)]),\ 0,\ [(Alice,\ 65),\ (Bob,\ 10)]))$
$(Rechtsfolge\ Erlaubnis)),$
$(Paragraph\ 11,$
$Rechtsnorm\ (Tatbestand\ ((0,\ [(Alice,\ 55),\ (Bob,\ 10)]),\ 0,\ [(Alice,\ 60),\ (Bob,\ 10)]))$
$(Rechtsfolge\ Erlaubnis)),$
$(Paragraph\ 10,$
$Rechtsnorm\ (Tatbestand\ ((0,\ [(Alice,\ 50),\ (Bob,\ 10)]),\ 0,\ [(Alice,\ 55),\ (Bob,\ 10)]))$
$(Rechtsfolge\ Erlaubnis)),$
$(Paragraph\ 9,$
$Rechtsnorm\ (Tatbestand\ ((2,\ [(Alice,\ 45),\ (Bob,\ 10)]),\ 0,\ [(Alice,\ 50),\ (Bob,\ 10)]))$
$(Rechtsfolge\ Erlaubnis)),$
$(Paragraph\ 8,$
$Rechtsnorm\ (Tatbestand\ ((7,\ [(Alice,\ 40),\ (Bob,\ 10)]),\ 2,\ [(Alice,\ 45),\ (Bob,\ 10)]))$
$(Rechtsfolge\ Erlaubnis)),$
$(Paragraph\ 7,$
$Rechtsnorm\ (Tatbestand\ ((12,\ [(Alice,\ 35),\ (Bob,\ 10)]),\ 7,\ [(Alice,\ 40),\ (Bob,\ 10)]))$
$(Rechtsfolge\ Erlaubnis)),$
$(Paragraph\ 6,$
$Rechtsnorm\ (Tatbestand\ ((17,\ [(Alice,\ 30),\ (Bob,\ 10)]),\ 12,\ [(Alice,\ 35),\ (Bob,\ 10)]))$
$(Rechtsfolge\ Erlaubnis)),$
$(Paragraph\ 5,$
$Rechtsnorm\ (Tatbestand\ ((22,\ [(Alice,\ 25),\ (Bob,\ 10)]),\ 17,\ [(Alice,\ 30),\ (Bob,\ 10)]))$
$(Rechtsfolge\ Erlaubnis)),$
$(Paragraph\ 4,$
$Rechtsnorm\ (Tatbestand\ ((27,\ [(Alice,\ 20),\ (Bob,\ 10)]),\ 22,\ [(Alice,\ 25),\ (Bob,\ 10)]))$
$(Rechtsfolge\ Erlaubnis)),$
$(Paragraph\ 3,$
$Rechtsnorm\ (Tatbestand\ ((32,\ [(Alice,\ 15),\ (Bob,\ 10)]),\ 27,\ [(Alice,\ 20),\ (Bob,\ 10)]))$
$(Rechtsfolge\ Erlaubnis)),$
$(Paragraph\ 2,$
$Rechtsnorm\ (Tatbestand\ ((37,\ [(Alice,\ 10),\ (Bob,\ 10)]),\ 32,\ [(Alice,\ 15),\ (Bob,\ 10)]))$
$(Rechtsfolge\ Erlaubnis)),$
$(Paragraph\ 1,$
$Rechtsnorm\ (Tatbestand\ ((42,\ [(Alice,\ 5),\ (Bob,\ 10)]),\ 37,\ [(Alice,\ 10),\ (Bob,\ 10)]))$
$(Rechtsfolge\ Erlaubnis))\}\rangle$

lemma ‹*beispiel-case-law'* (*HandlungF* (*abbauen 5*)) =
Gesetz
{(*Paragraph 1*, *Rechtsnorm* (*Tatbestand* [*Gewinnt Alice 5*]) (*Rechtsfolge Erlaubnis*))}›

Helper

definition *floor* :: *real* ⇒ *nat* **where**
floor x ≡ *nat* ⌊*x*⌋

lemma *floorD*: *a* ≤ *b* ⟹ *floor a* ≤ *floor b*

lemma *floor-minusD*:
fixes *a* :: *nat* **and** *a'* :: *real*
shows *a* ≤ *b* ⟹ *a* − *a'* ≤ *b* − *b'* ⟹ *a* − *floor a'* ≤ *b* − *floor b'*

10 Experiment: Steuergesetzgebung

Basierend auf einer stark vereinfachten Version des deutschen Steuerrechts. Wenn ich Wikipedia richtig verstanden habe, habe ich sogar aus Versehen einen Teil des österreichischen Steuersystem gebaut mit deutschen Konstanten.

locale *steuer-defs* =
fixes *steuer* :: *nat* ⇒ *nat* — Einkommen -> Steuer
begin
definition *brutto* :: *nat* ⇒ *nat* **where**
brutto einkommen ≡ *einkommen*
definition *netto* :: *nat* ⇒ *nat* **where**
netto einkommen ≡ *einkommen* − (*steuer einkommen*)
definition *steuersatz* :: *nat* ⇒ *percentage* **where**
steuersatz einkommen ≡ *percentage* ((*steuer einkommen*) / *einkommen*)
end

Beispiel

definition *beispiel-25prozent-steuer* :: *nat* ⇒ *nat* **where**
beispiel-25prozent-steuer e ≡ *nat* ⌊*real e* ∗ (*percentage 0.25*)⌋

lemma *beispiel-25prozent-steuer 100 = 25*
steuer-defs.brutto 100 = 100
steuer-defs.netto beispiel-25prozent-steuer 100 = 75
steuer-defs.steuersatz beispiel-25prozent-steuer 100 = percentage 0.25

lemma *steuer-defs.steuersatz beispiel-25prozent-steuer 103 = percentage* (*25 / 103*)
percentage (*25 / 103*) ≤ *percentage 0.25*
(*103*::*nat*) > *100*

locale *steuersystem* = *steuer-defs* +
  assumes *wer-hat-der-gibt*:
    einkommen-a ≥ *einkommen-b* ⟹ *steuer einkommen-a* ≥ *steuer einkommen-b*

  and *leistung-lohnt-sich*:
    einkommen-a ≥ *einkommen-b* ⟹ *netto einkommen-a* ≥ *netto einkommen-b*

  — Ein Existenzminimum wird nicht versteuert. Zahl Deutschland 2022, vermutlich sogar die falsche Zahl.
  and *existenzminimum*:
    einkommen ≤ *9888* ⟹ *steuer einkommen* = *0*

begin

end

fun *zonensteuer* :: (*nat* × *percentage*) *list* ⇒ *percentage* ⇒ *nat* ⇒ *real* **where**
   zonensteuer ((*zone*, *prozent*)#*zonen*) *spitzensteuer e* =
       ((*min zone e*) ∗ *prozent*) + (*zonensteuer zonen spitzensteuer* (*e* − *zone*))
| *zonensteuer* [] *spitzensteuer e* = *e*∗*spitzensteuer*

lemma *zonensteuermono*: *e1* ≤ *e2*
  ⟹ *zonensteuer zs spitzensteuer e1* ≤ *zonensteuer zs spitzensteuer e2*

Kein Einkommen -> keine Steuer

lemma *zonensteuer-zero*: *zonensteuer ls p 0* = *0*

Steuer ist immer positiv.

lemma *zonensteuer-pos*: *zonensteuer ls p e* ≥ *0*

Steuer kann nicht höher sein als das Einkommen.

lemma *zonensteuer-limit*: *zonensteuer ls spitzensteuer einkommen* ≤ *einkommen*

lemma *zonensteuer-leistung-lohnt-sich*: *e1* ≤ *e2*
  ⟹ *e1* − *zonensteuer zs spitzensteuer e1* ≤ *e2* − *zonensteuer zs spitzensteuer e2*

definition *steuerzonen2022* :: (*nat* × *percentage*) *list* **where**
  steuerzonen2022 ≡ [
                    (*10347*, *percentage 0*),
                    (*4579*, *percentage 0.14*),
                    (*43670*, *percentage 0.2397*),
                    (*219229*, *percentage 0.42*)
                   ]

```
fun steuerzonenAbs :: (nat × percentage) list ⇒ (nat × percentage) list where
   steuerzonenAbs [] = []
 | steuerzonenAbs ((zone, prozent)#zonen) =
     (zone,prozent)#(map (λ(z,p). (zone+z, p)) (steuerzonenAbs zonen))

definition steuerbuckets2022 :: (nat × percentage) list where
  steuerbuckets2022 ≡ [
                       (10347, percentage 0),
                       (14926, percentage 0.14),
                       (58596, percentage 0.2397),
                       (277825, percentage 0.42)
                      ]


lemma steuerbuckets2022: steuerbuckets2022 = steuerzonenAbs steuerzonen2022

fun wfSteuerbuckets :: (nat × percentage) list ⇒ bool where
  wfSteuerbuckets [] = True
| wfSteuerbuckets [bs] = True
| wfSteuerbuckets ((b1, p1)#(b2, p2)#bs) ⟷ b1 ≤ b2 ∧ wfSteuerbuckets ((b2,p2)#bs)


fun bucketsteuerAbs :: (nat × percentage) list ⇒ percentage ⇒ nat ⇒ real where
   bucketsteuerAbs ((bis, prozent)#mehr) spitzensteuer e =
       ((min bis e) * prozent)
       + (bucketsteuerAbs (map (λ(s,p). (s−bis,p)) mehr) spitzensteuer (e − bis))
|  bucketsteuerAbs [] spitzensteuer e = e*spitzensteuer

lemma wfSteuerbucketsConsD: wfSteuerbuckets (z#zs) ⟹ wfSteuerbuckets zs

lemma wfSteuerbucketsMapD:
  wfSteuerbuckets (map (λ(z, y). (zone + z, y)) zs) ⟹  wfSteuerbuckets zs


lemma mapHelp1: wfSteuerbuckets zs ⟹
      (map ((λ(s, y). (s − x, y)) ∘ (λ(z, y). (x + z, y)))) zs = zs

lemma bucketsteuerAbs-zonensteuer:
  wfSteuerbuckets (steuerzonenAbs zs) ⟹
      bucketsteuerAbs (steuerzonenAbs zs) spitzensteuer e
      = zonensteuer zs spitzensteuer e


definition einkommenssteuer :: nat ⇒ nat where
  einkommenssteuer einkommen ≡
    floor (bucketsteuerAbs steuerbuckets2022 (percentage  0.45) einkommen)

value ‹einkommenssteuer 10›
```

lemma ‹*einkommenssteuer 10 = 0*›
lemma ‹*einkommenssteuer 10000 = 0*›
lemma ‹*einkommenssteuer 14000 = floor ((14000−10347)∗0.14)*›
lemma ‹*einkommenssteuer 20000 =*
floor ((14926−10347)∗0.14 + (20000−14926)∗0.2397)›
value ‹*einkommenssteuer 40000*›
value ‹*einkommenssteuer 60000*›

lemma *einkommenssteuer*:
einkommenssteuer einkommen =
floor (zonensteuer steuerzonen2022 (percentage 0.45) einkommen)

interpretation *steuersystem*
where *steuer = einkommenssteuer*

11 Beispiel: Steuern

Wenn die Welt sich durch eine Zahl darstellen lässt, ...

datatype *steuerwelt = Steuerwelt*
(*get-einkommen*: *person ⇒ int*) — einkommen: einkommen jeder Person (im Zweifel 0).

fun *steuerlast* :: *person ⇒ steuerwelt handlung ⇒ int* **where**
steuerlast p (Handlung vor nach) = ((get-einkommen vor) p) − ((get-einkommen nach) p)

fun *brutto* :: *person ⇒ steuerwelt handlung ⇒ int* **where**
brutto p (Handlung vor nach) = (get-einkommen vor) p
fun *netto* :: *person ⇒ steuerwelt handlung ⇒ int* **where**
netto p (Handlung vor nach) = (get-einkommen nach) p

Default: kein Einkommen. Um Beispiele einfacher zu schreiben.

definition *KE* :: *person ⇒ int* **where**
KE ≡ λp. 0

lemma ‹*steuerlast Alice (Handlung (Steuerwelt (KE(Alice:=8))) (Steuerwelt (KE(Alice:=5)))) = 3*›
lemma ‹*steuerlast Alice (Handlung (Steuerwelt (KE(Alice:=8))) (Steuerwelt (KE(Alice:=0)))) = 8*›
lemma ‹*steuerlast Bob (Handlung (Steuerwelt (KE(Alice:=8))) (Steuerwelt (KE(Alice:=5)))) = 0*›
lemma ‹*steuerlast Alice (Handlung (Steuerwelt (KE(Alice:=−3))) (Steuerwelt (KE(Alice:=−4)))) = 1*›
lemma ‹*steuerlast Alice (Handlung (Steuerwelt (KE(Alice:=1))) (Steuerwelt (KE(Alice:=−1)))) = 2*›

fun *mehrverdiener* :: *person ⇒ steuerwelt handlung ⇒ person set* **where**
mehrverdiener ich (Handlung vor nach) = {p. (get-einkommen vor) p ≥ (get-einkommen vor) ich}

lemma ‹*mehrverdiener Alice*
(Handlung (Steuerwelt (KE(Alice:=8, Bob:=12, Eve:=7))) (Steuerwelt (KE(Alice:=5))))
= {Alice, Bob}›

```
definition maxime-steuern :: (person, steuerwelt) maxime where
  maxime-steuern ≡ Maxime
      (λich handlung.
          (∀p∈mehrverdiener ich handlung.
              steuerlast ich handlung ≤ steuerlast p handlung)
        ∧ (∀p∈mehrverdiener ich handlung.
              netto ich handlung ≤ netto p handlung)
        )



fun delta-steuerwelt :: (steuerwelt, person, int) delta where
  delta-steuerwelt (Handlung vor nach) =
      Aenderung.delta-num-fun (Handlung (get-einkommen vor) (get-einkommen nach))

definition sc ≡ SimConsts
    Alice
    maxime-steuern
    (printable-case-law-ableiten-absolut (λw. show-fun (get-einkommen w)))
definition sc' ≡ SimConsts
    Alice
    maxime-steuern
    (case-law-ableiten-relativ delta-steuerwelt)

definition initialwelt ≡ Steuerwelt (KE(Alice:=8, Bob:=3, Eve:= 5))

definition beispiel-case-law h ≡ simulateOne sc 3 h initialwelt (Gesetz {})
definition beispiel-case-law' h ≡ simulateOne sc' 20 h initialwelt (Gesetz {})
```

Keiner zahlt steuern: funktioniert

```
value ‹beispiel-case-law (HandlungF (λich welt. welt))›
lemma ‹beispiel-case-law' (HandlungF (λich welt. welt)) =
  Gesetz {(Paragraph 1, Rechtsnorm (Tatbestand []) (Rechtsfolge Erlaubnis))}›
```

Ich zahle 1 Steuer: funnktioniert nicht, ... komisch, sollte aber? Achjaaaaaa, jeder muss ja Steuer zahlen, ...

```
definition ich-zahle-1-steuer ich welt ≡
  Steuerwelt ((get-einkommen welt)(ich := ((get-einkommen welt) ich) − 1))
lemma ‹beispiel-case-law (HandlungF ich-zahle-1-steuer) =
  Gesetz
  {(Paragraph 1,
    Rechtsnorm
     (Tatbestand
       ([(Alice, 8), (Bob, 3), (Carol, 0), (Eve, 5)],
        [(Alice, 7), (Bob, 3), (Carol, 0), (Eve, 5)]))
     (Rechtsfolge Verbot))}›
lemma ‹beispiel-case-law' (HandlungF ich-zahle-1-steuer) =
  Gesetz
```

```
  {(Paragraph 1, Rechtsnorm (Tatbestand [Verliert Alice 1])
                            (Rechtsfolge Verbot))}›
```

Jeder muss steuern zahlen: funktioniert, ist aber doof, denn am Ende sind alle im Minus.

Das *ich* wird garnicht verwendet, da jeder Steuern zahlt.

```
definition jeder-zahle-1-steuer ich welt ≡
  Steuerwelt ((λe. e − 1) ∘ (get-einkommen welt))
lemma ‹beispiel-case-law (HandlungF jeder-zahle-1-steuer) =
Gesetz
  {(Paragraph 3,
    Rechtsnorm
     (Tatbestand
       ([(Alice, 6), (Bob, 1), (Carol, − 2), (Eve, 3)],
        [(Alice, 5), (Bob, 0), (Carol, − 3), (Eve, 2)]))
     (Rechtsfolge Erlaubnis)),
   (Paragraph 2,
    Rechtsnorm
     (Tatbestand
       ([(Alice, 7), (Bob, 2), (Carol, − 1), (Eve, 4)],
        [(Alice, 6), (Bob, 1), (Carol, − 2), (Eve, 3)]))
     (Rechtsfolge Erlaubnis)),
   (Paragraph 1,
    Rechtsnorm
     (Tatbestand
       ([(Alice, 8), (Bob, 3), (Carol, 0), (Eve, 5)],
        [(Alice, 7), (Bob, 2), (Carol, − 1), (Eve, 4)]))
     (Rechtsfolge Erlaubnis))}›
lemma ‹beispiel-case-law' (HandlungF jeder-zahle-1-steuer) =
  Gesetz
  {(Paragraph 1,
    Rechtsnorm
     (Tatbestand [Verliert Alice 1, Verliert Bob 1, Verliert Carol 1, Verliert Eve 1])
     (Rechtsfolge Erlaubnis))}›
```

Jetzt kommt die Steuern.thy ins Spiel.

Bei dem geringen Einkommen zahlt keiner Steuern.

```
definition jeder-zahlt steuerberechnung ich welt ≡
  Steuerwelt ((λe. e − steuerberechnung e) ∘ nat ∘ (get-einkommen welt))

definition jeder-zahlt-einkommenssteuer ≡ jeder-zahlt einkommenssteuer
lemma ‹beispiel-case-law (HandlungF jeder-zahlt-einkommenssteuer ) =
  Gesetz
  {(Paragraph 1,
    Rechtsnorm
     (Tatbestand
       ([(Alice, 8), (Bob, 3), (Carol, 0), (Eve, 5)],
        [(Alice, 8), (Bob, 3), (Carol, 0), (Eve, 5)]))
     (Rechtsfolge Erlaubnis))}›
```

lemma ‹*simulateOne*
sc' 1
(*HandlungF jeder-zahlt-einkommenssteuer*)
(*Steuerwelt* (*KE*(*Alice*:=*10000*, *Bob*:=*14000*, *Eve*:= *20000*)))
(*Gesetz* {})
=
Gesetz
{(*Paragraph 1*,
Rechtsnorm (*Tatbestand* [*Verliert Bob 511*, *Verliert Eve 1857*])
(*Rechtsfolge Erlaubnis*))}›

Die Anforderungen fuer ein *steuersystem* und die *maxime-steuern* sind vereinbar.

lemma *steuersystem steuersystem-impl* ⟹
(∀ *welt*. *teste-maxime welt* (*HandlungF* (*jeder-zahlt steuersystem-impl*)) *maxime-steuern*)

lemma $a \leq x \Longrightarrow int\ x - int\ (x - a) = a$

Danke ihr nats. Macht also keinen Sinn das als Annahme in die Maxime zu packen....

lemma *steuern-kleiner-einkommen-nat*:
steuerlast ich (*Handlung welt* (*jeder-zahlt steuersystem-impl ich welt*))
≤ *brutto ich* (*Handlung welt* (*jeder-zahlt steuersystem-impl ich welt*))

lemma (∀ *einkommen*. *steuersystem-impl einkommen* ≤ *einkommen*) ⟹
(∀ *einkommen*. *einkommen* ≤ *9888* ⟶ *steuersystem-impl einkommen* = *0*) ⟹
∀ *welt*. *teste-maxime welt* (*HandlungF* (*jeder-zahlt steuersystem-impl*)) *maxime-steuern*
⟹ *steuersystem steuersystem-impl*