Exercise 12.1 Termination for sign analysis

Recall the abstract interpreter from the last sheet:

```plaintext
datatype sign = Pos | Zero | Neg | Any

instantiation sign :: semilattice sup top
begin
  definition less_eq_sign where "x ≤ y = (y = Any ∨ x=y)"
  definition less_sign where "x < y = (x ≤ y ∧ ¬ y ≤ (x::sign))"
  definition sup_sign where "x ⊔ y = (if x = y then x else Any)"
  definition top_sign where "⊤ = Any"

instance by standard (auto simp: less_eq_sign_def less_sign_def sup_sign_def top_sign_def)
end

fun γ_sign :: "sign ⇒ val set" where
  "γ_sign Neg = {i. i < 0}" |
  "γ_sign Pos = {i. i > 0}" |
  "γ_sign Zero = {0}" |
  "γ_sign Any = UNIV"

fun num_sign :: "val ⇒ sign" where
  "num_sign i = (if i = 0 then Zero else if i > 0 then Pos else Neg)"

fun plus_sign :: "sign ⇒ sign ⇒ sign" where
  "plus_sign y Zero = y" |
  "plus_sign Zero y = y" |
  "plus_sign x y = (if x = y then x else Any)"

global_interpretation Val_semilattice
  where γ = γ_sign and num' = num_sign and plus' = plus_sign
```

Semantics of Programming Languages

Exercise Sheet 12
proof (standard, goal_cases)
  case (4 a1 a2) thus ?case
    by (induction a1 a2 rule: plus.induct) (auto simp add:mod_add_eq)
qed (auto simp: less_eq_sign_def top_sign_def)

global_interpretation Abs_Int
  where γ = γ.sign and num' = num.sign and plus' = plus.sign
  defines aval.sign = aval' and step.sign = step' and AI.sign = AI ..

Define a measure function on the abstract domain, which can be used to prove that the analysis always terminates. Define a function m.sign from the sign domain into the natural numbers such that

- \( x < y \implies m.sign x > m.sign y \)
- \( m.sign x \leq h.sign \)

where \( h.sign \) is the height of the sign domain.

abbreviation h.sign :: nat
fun m.sign :: “sign ⇒ nat”

global_interpretation Abs_Int_mono
  where γ = γ.sign and num' = num.sign and plus' = plus.sign

global_interpretation Abs_Int_measure
  where γ = γ.sign and num' = num.sign and plus' = plus.sign
  and m = m.sign and h = h.sign

Exercise 12.2 Inverse Analysis

Consider a similar analysis based on this abstract domain:

datatype sign0 = None | Neg | Pos0 | Any

fun γ.0 :: “sign0 ⇒ val set” where
  “γ.0 None = {}” |
  “γ.0 Neg = { i. i < 0 }” |
  “γ.0 Pos0 = { i. i ≥ 0 }” |
  “γ.0 Any = UNIV”

Define inverse analyses for “+” and “<” and prove the required correctness properties:

fun inv.plus' :: “sign0 ⇒ sign0 ⇒ sign0 ⇒ sign0 * sign0”
lemma “[ inv.plus' a1 a2 (a1',a2'); i1 ∈ γ.0 a1; i2 ∈ γ.0 a2; i1+i2 ∈ γ.0 a ]
  ⇒ i1 ∈ γ.0 a1' ∧ i2 ∈ γ.0 a2’”

fun inv.less' :: “bool ⇒ sign0 ⇒ sign0 ⇒ sign0 * sign0”
lemma “[ inv.less' bv a1 a2 (a1',a2'); i1 ∈ γ.0 a1; i2 ∈ γ.0 a2; (i1<i2) = bv ]
  ⇒ i1 ∈ γ.0 a1' ∧ i2 ∈ γ.0 a2’”
Homework 12.1  Al Table

Submission until Sunday, Feb 7, 23:59.
Consider the following IMP program:

```plaintext
r := 11;
a := 11 + 11;
WHILE b DO ( 
r := r + 1;
a := a - 2
);
r := a + 1
```

Add annotations for parity analysis to this program, and iterate on it the \( \text{step}' \) function until a fixed point is reached. (More precisely, let \( C \) be the annotated program; you need to compute \( (\text{step}' \top)^0 C, (\text{step}' \top)^1 C, (\text{step}' \top)^2 C, \) etc.). Document the results of each iteration in a table. For brevity, only write down changed values, and denote \( x, y \) for \( \{r:=x,a:=y\} \).

Homework 12.2  Al for finite words

Submission until Sunday, Feb 7, 23:59.
We change the language of arithmetic expression in IMP to bitwise arithmetic on 4-bit words. First, we define a type \textit{word} that holds precisely four elements. We can instantiate this with \textit{bool} to obtain a type for 4-bit words.

```plaintext
datatype 'a word = Word 'a 'a 'a 'a
type synonym vname = string
type synonym val = "bool word"
type synonym state = "vname \Rightarrow val"
datatype aexp = N val | V vname | Bit\_And aexp aexp | Bit\_Or aexp aexp
```

The abstract interpretation framework is already set up for this IMP variant.
Your task is to define abstract interpretation that assigns each bit in a word \textit{True}, \textit{False}, either, or none.

```plaintext
datatype parity = T | F | Either | None
```

First, instantiate the abstract interpreter with termination:

```plaintext
fun \_\_parity
fun conj\_parity
fun disj\_parity
fun num\_parity
instantiation parity :: "\{order, semilattice\_sup\_top, bounded\_lattice\}"
```

3
begin

definition less_eq_parity
definition less_parity
definition sup_parity
definition inf_parity
definition top_parity
definition bot_parity
instance
end

type synonym word_parity = “parity word”

fun γ_word_parity :: “word_parity ⇒ val set”
definition and_parity :: “word_parity ⇒ word_parity ⇒ word_parity”
definition or_parity :: “word_parity ⇒ word_parity ⇒ word_parity”
definition num_word_parity :: “val ⇒ word_parity”
global_interpretation Val_semilattice
  where γ = γ_word_parity and num’ = num_word_parity and and’ = and_parity and or’ = or_parity

global_interpretation Abs_Int
  where γ = γ_word_parity and num’ = num_word_parity and and’ = and_parity and or’ = or_parity
  defines step_parity = step’ and AI_parity = AI

global_interpretation Abs_Int_mono
where γ = γ_word_parity and num’ = num_word_parity and and’ = and_parity and or’ = or_parity

proof (standard, goal_cases)

Then, instantiate the inverse analysis framework:

global_interpretation Val_lattice_gamma
  where γ = γ_word_parity and num’ = num_word_parity and and’ = and_parity and or’ = or_parity
definition test_num_word_parity :: “val ⇒ word_parity ⇒ bool”
definition inv_and_word_parity ::
  “word_parity ⇒ word_parity ⇒ word_parity ⇒ (word_parity × word_parity)”
definition inv_or_word_parity ::
  “word_parity ⇒ word_parity ⇒ word_parity ⇒ (word_parity × word_parity)”

Your inverse analysis of the less may be rather approximative, but not trivial.
For a more precise analysis, up to three bonus points are awarded.
definition inv_less_word_parity ::
  “bool ⇒ word_parity ⇒ word_parity ⇒ (word_parity × word_parity)”

global_interpretation Abs_Int_inv
where γ = γ_word_parity and num’ = num_word_parity and and’ = and_parity and or’ = or_parity
and test_num’ = test_num_word_parity and inv_and’ = inv_and_word_parity
and $ inv \_or' = inv \_or \_word \_parity$ and $ inv \_less' = inv \_less \_word \_parity$

defines $ step \_parity' = step' \ and \ AI \_parity' = AI'$