Technische Universität München
Fakultät für Informatik
Prof. Tobias Nipkow, Ph.D.
Fabian Huch

WS 2020/21
1.2.2021

Semantics of Programming Languages

Exercise Sheet 12

Exercise 12.1 Termination for sign analysis

Recall the abstract interpreter from the last sheet:

datatype *sign* = *Pos* | *Zero* | *Neg* | *Any*

instantiation *sign* :: *semilattice_sup_top*
begin

definition *less_eq_sign* **where** "$x \le y = (y = Any \lor x{=}y)$"

definition *less_sign* **where** "$x < y = (x \le y \land \neg\ y \le (x{::}sign))$"

definition *sup_sign* **where** "$x \sqcup y = (if\ x = y\ then\ x\ else\ Any)$"

definition *top_sign* **where** "$\top = Any$"

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**
"$\gamma\_sign\ Neg = \{i.\ i < 0\}$" |
"$\gamma\_sign\ Pos\ = \{i.\ i > 0\}$" |
"$\gamma\_sign\ Zero\ = \{0\}$" |
"$\gamma\_sign\ Any = UNIV$"

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 $\gamma = \gamma\_sign$ **and** $num' = num\_sign$ **and** $plus' = plus\_sign$

proof (*standard*, *goal_cases*)
  case (*4 _ a1 _ a2*) **thus** *?case*
    by (*induction a1 a2 rule*: *plus_sign.induct*) (*auto simp add*:*mod_add_eq*)
qed (*auto simp*: *less_eq_sign_def top_sign_def*)

global_interpretation *Abs_Int*
  where $\gamma = \gamma\_sign$ **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

- $x < y \Longrightarrow m\_sign\ x > m\_sign\ y$
- $m\_sign\ x \le h\_sign$

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 $\gamma = \gamma\_sign$ **and** $num' = num\_sign$ **and** $plus' = plus\_sign$
global_interpretation *Abs_Int_measure*
where $\gamma = \gamma\_sign$ **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**
"$\gamma\_0\ None = \{\}$" |
"$\gamma\_0\ Neg = \{i.\ i < 0\}$" |
"$\gamma\_0\ Pos0 = \{i.\ i \ge 0\}$" |
"$\gamma\_0\ Any = UNIV$"

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

fun *inv_plus'* :: *"sign0 ⇒ sign0 ⇒ sign0 ⇒ sign0 ∗ sign0"*
lemma
  "$[\![\ inv\_plus'\ a\ a1\ a2 = (a1',a2');\ \ i1 \in \gamma\_0\ a1;\ \ i2 \in \gamma\_0\ a2;\ i1{+}i2 \in \gamma\_0\ a\ ]\!]$
  $\Longrightarrow i1 \in \gamma\_0\ a1' \wedge i2 \in \gamma\_0\ a2'$ "
fun *inv_less'* :: *"bool ⇒ sign0 ⇒ sign0 ⇒ sign0 ∗ sign0"*
lemma
  "$[\![\ inv\_less'\ bv\ a1\ a2 = (a1',a2');\ \ i1 \in \gamma\_0\ a1;\ \ i2 \in \gamma\_0\ a2;\ (i1{<}i2) = bv\ ]\!]$
  $\Longrightarrow i1 \in \gamma\_0\ a1' \wedge i2 \in \gamma\_0\ a2'$"

Homework 12.1 AI Table

Submission until Sunday, Feb 7, 23:59.

Consider the following IMP program:

```
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 *step′* function until a fixed point is reached. (More precisely, let C be the annotated program; you need to compute $(step'\ \top)^0\ C$, $(step'\ \top)^1\ C$, $(step'\ \top)^2\ C$, 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 AI 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 *word* that holds precisely four elements. We can instantiate this with *bool* to obtain a type for 4-bit words.

datatype *'a word = Word 'a 'a 'a 'a*

type_synonym *vname = string*
type_synonym *val = "bool word"*
type_synonym *state = "vname ⇒ 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 *True*, *False*, either, or none.

datatype *parity = T | F | Either | None*

First, instantiate the abstract interpreter with termination:

fun *γ_parity*
fun *conj_parity*
fun *disj_parity*
fun *num_parity*
instantiation *parity :: "{order, semilattice_sup_top, bounded_lattice}"*

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** $AI\_parity' = AI'$