Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Fabian Huch, Kevin Kappelmann

WS 23/24
30.01.2024

Semantics of Programming Languages

Exercise Sheet 13

Exercise 13.1 Complete Lattices

Which of the following ordered sets are complete lattices?

- $\mathbb{N}$, the set of natural numbers $\{0, 1, 2, 3, \ldots\}$ with the usual order
- $\mathbb{N} \cup \{\infty\}$, the set of natural numbers plus infinity, with the usual order and $n < \infty$ for all $n \in \mathbb{N}$.
- A finite set A with a total order $\leq$ on it.

Exercise 13.2 Sign Analysis

Instantiate the abstract interpretation framework to a sign analysis over the lattice $pos, zero, neg, any$, where pos abstracts positive values, $zero$ abstracts zero, neg abstracts negative values, and any abstracts any value.

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

instantiation *sign :: order*
instantiation *sign :: semilattice_sup_top*
fun *γ_sign :: "sign ⇒ val set"*
fun *num_sign :: "val ⇒ sign"*
fun *plus_sign :: "sign ⇒ sign ⇒ sign"*
global_interpretation *Val_semilattice*
  where *γ = γ_sign* **and** *num′ = num_sign* **and** *plus′ = plus_sign*
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*

Some tests:

definition *"test1_sign =*
  ''x'' ::= N 1;;
  WHILE Less (V ''x'') (N 100) DO ''x'' ::= Plus (V ''x'') (N 2)"
value *"show_acom (the(AI_sign test1_sign))"*

definition *"test2_sign =*
  ''x'' ::= N 1;;

WHILE Less (V ''x'') (N 100) DO ''x'' ::= Plus (V ''x'') (N 3)"

definition *"steps c i = ((step_sign ⊤) ^^ i) (bot c)"*

value *"show_acom (steps test2_sign 0)"*

...

value *"show_acom (steps test2_sign 6)"*
value *"show_acom (the(AI_sign test2_sign))"*

Exercise 13.3 AI for Conditionals

Our current constant analysis does not regard conditionals. For example, it cannot figure out, that after executing the program *x*:=*2*; *IF x*<*2 THEN x*:=*2 ELSE x*:=*1*, *x* will be constant.

In this exercise, we extend our abstract interpreter with a simple analysis of boolean expressions. To this end, modify locale *Val_semilattice* in theory *Abs_Int0.thy* as follows:

- Introduce an abstract domain *'bv* for boolean values, add, analogously to *num'* and *plus'* also functions for the boolean operations and for *less*.
- Modify *Abs_Int0* to accommodate for your changes.

Homework 13.1 Bits analysis

Submission until Monday, Feb 5, 23:59pm.

An interesting analysis for abstract interpretation is whether a program could have an arithmetic overflow. For this analysis, we consider the abstract domain of a bounded number of bits. Additionally, we store the sign information. We also have cases for zero and any (though *0* is also contained in *Pos* and *Neg*):

datatype *bits = Zero | Pos nat | Neg nat | Either nat | Any*

The constructors *Pos*, *Neg*, and *Either* take a natural number *b* and represent all corresponding numbers whose binary representation needs at most *b+1* bits:

$\gamma\_bits\ Any = UNIV$

$\gamma\_bits\ Zero = \{0\}$

$\gamma\_bits\ (Pos\ b) = \{i.\ 0 \le i \wedge |i| < 2^{Suc\ b}\}$

$\gamma\_bits\ (Neg\ b) = \{i.\ i \le 0 \wedge |i| < 2^{Suc\ b}\}$

$\gamma\_bits\ (Either\ b) = \{i.\ |i| < 2^{Suc\ b}\}$

Instantiate the *order* and *semilattice_sup_top* classes such that they are suitable for abstract interpretation. Be as precise as possible!

instantiation *bits* :: *order*
instantiation *bits* :: *semilattice_sup_top*

Next, define the abstraction function. It must be executable.

Hint:

- do not use *log2* (since that works over reals), instead define your own function.

fun *num_bits* :: *"val ⇒ bits"*

Some tests:

value *"num_bits 0 = Zero"*
value *"num_bits 3 = Pos 1"*
value *"num_bits (−42) = Neg 5"*

Next, we want to instantiate the abstract interpreter. As the analysis depends on the exact size of the machine words, we introduce a locale with a single parameter *bits* for the number of bits, and use the *sublocale* command instead of *global_interpretation*.

Background: Sublocale makes our locale extend the abstract interpretation locales. In particular, any concept defined in the abstract interpretation locales will be available in our locale as well. Once we instantiate bounded_bits for a concrete number of bits, we can use the abstract interpreter.

locale *bounded_bits* = **fixes** *bits* :: *nat*
begin

Define the abstract plus operation (be as precise as possible) and complete the instance proofs. While there is no explicit assumption, in your construction you may assume that all input values of the *bits* type are bounded by the *bits* variable. Your output must also be bounded.

Hint:

- The number of subgoals that arise in this construction can be quite large. But if you do things right, they should mostly be easy, so you can solve even hundreds of subgoals with a single *auto* call (which may take a few seconds to complete).

fun *plus_bits* :: *"bits ⇒ bits ⇒ bits"*
sublocale *Val_semilattice*
  where $\gamma = \gamma$*_bits* **and** *num′* = *num_bits* **and** *plus′* = *plus_bits*
sublocale *Abs_Int*
  where $\gamma = \gamma$*_bits* **and** *num′* = *num_bits* **and** *plus′* = *plus_bits*
sublocale *Abs_Int_mono*
  where $\gamma = \gamma$*_bits* **and** *num′* = *num_bits* **and** *plus′* = *plus_bits*
end

Finally, an example for 4-bit machine words (try your own!):

global_interpretation *bounded_bits*
  where *bits="Suc (Suc (Suc 0))"*
  defines *AI_bits4* = *AI* **and** *step_bits4* = *step′* **and** *aval′_bits4* = *aval′*
  done

definition *"steps c i = (step_bits4* ⊤ ^^ *i) (Abs_Int0.bot c)"*

definition *"test1_bits =*
''y'' ::= N 7;;
''z'' ::= Plus (V ''y'') (N 2);;
''y'' ::= Plus (V ''x'') (N 0)"

value *"show_acom (steps test1_bits 0)"*
value *"show_acom (steps test1_bits 1)"*
value *"show_acom (steps test1_bits 2)"*
value *"show_acom (steps test1_bits 3)"*

value *"show_acom (the (AI_bits4 test1_bits))"*