theory *EX-DoubleAddition = SALOverflowFWInst* :

1 Example - Double Addition

We verify that a program computing four times x does not overflow. The annotation at the HALT instruction is not necessary for the safety proof, but it demonstrates that we can also do correctness proofs in our system.

1.1 Variables

The program computes 4 times x.

constdefs *x*::*nat*
x≡*1*

1.2 Annotated Program

We initialize x with 3 (any other value v such that 4*v ¡= MAX could also be used) do two successive additions on it. The annoation in the end says that the result is NAT 12.

constdefs *prog*::*SALprogram*
prog ≡ [(*0*,[
(*SET x 3*,*None*),
(*ADD x x*,*None*),
(*ADD x x*,*None*),
(*HALT*, *Some* (λ (*pc*,*m*,*e*). *m x* = (*NAT 12*)))])]

1.3 Verification Condition

We generate an executable ML program for the VCG

generate-code (*vcgSAL.ML*) [*term-of*]
vcg = *vcgSAL*
prg = *prog*

This is the result this executable VCG gives us for the example program above. The formula guaranees that the postcondition is valid when we reach HALT and that all instructions in between are type safe and do not overflow

constdefs *vc*::*SALform*
vc ≡ %*s*. (%*s*. (%(*pc*, *m*, *e*). *pc* = (*0*::*nat*, *0*::*nat*) & *env.cs e* = [(*0*::*nat*, *m*)] & *env.h e* = [] & (*ALL X*. *m X* = *ILLEGAL*)) *s* −−> (%*s*. (%(*pc*, *m*, *e*). *Suc* (*Suc* (*Suc* (*0*::*nat*))) <= *MAX*) *s* & (%*s*. (%*s*. (%(*pc*, *m*, *e*). *pc* = (*0*::*nat*, *0*::*nat*)) *s* −−> (%(*pc*, *m*, *e*). (%*s*. (%(*pc*, *m*, *e*). (*EX n*. *m* (*Suc* (*0*::*nat*)) = *NAT n*) & (*EX n*. *m* (*Suc* (*0*::*nat*)) = *NAT n*) & (*EX n*. *lift op* + (*m* (*Suc* (*0*::*nat*))) (*m* (*Suc* (*0*::*nat*))) = *NAT n* & *n* <= *MAX*)) *s* & (%*s*. (%*s*. (%(*pc*, *m*, *e*). *pc* = (*0*::*nat*, *Suc* (*0*::*nat*))) *s* −−> (%(*pc*, *m*, *e*). (%*s*. (%(*pc*, *m*, *e*). (*EX n*. *m* (*Suc* (*0*::*nat*))

$= NAT\ n)\ \&\ (EX\ n.\ m\ (Suc\ (0{::}nat)) = NAT\ n)\ \&\ (EX\ n.\ lift\ op\ +\ (m\ (Suc\ (0{::}nat)))\ (m\ (Suc\ (0{::}nat))) = NAT\ n\ \&\ n <= MAX))\ s\ \&\ (\%s.\ (\%(pc,\ m,\ e).\ pc = (0{::}nat,\ Suc\ (Suc\ (0{::}nat))))\ s\ --{>}\ (\%(pc,\ m,\ e).\ (\%s.\ (\%s.\ True)\ s\ \&\ (\%s.\ (\%(pc,\ m,\ e).\ m\ x = NAT\ (12{::}nat))\ s\ \&\ (\%s.\ True)\ s)\ s)\ ((0{::}nat,\ Suc\ (Suc\ (Suc\ (0{::}nat)))),\ update\ m\ (Suc\ (0{::}nat))\ (lift\ op\ +\ (m\ (Suc\ (0{::}nat)))\ (m\ (Suc\ (0{::}nat)))),\ env.h\text{-}update\ (env.h\ e\ @\ [(0{::}nat,\ Suc\ (Suc\ (0{::}nat)))])\ e))\ s)\ s\ \&\ (\%s.\ True)\ s)\ s)\ ((0{::}nat,\ Suc\ (Suc\ (0{::}nat))),\ update\ m\ (Suc\ (0{::}nat))\ (lift\ op\ +\ (m\ (Suc\ (0{::}nat)))\ (m\ (Suc\ (0{::}nat)))),\ env.h\text{-}update\ (env.h\ e\ @\ [(0{::}nat,\ Suc\ (0{::}nat))])\ e))\ s)\ s\ \&\ (\%s.\ True)\ s)\ s)\ ((0{::}nat,\ Suc\ (0{::}nat)),\ update\ m\ (Suc\ (0{::}nat))\ (NAT\ (Suc\ (Suc\ (Suc\ (0{::}nat))))),\ env.h\text{-}update\ (env.h\ e\ @\ [(0{::}nat,\ 0{::}nat)])\ e))\ s)\ s\ \&\ (\%s.\ True)\ s)\ s)\ s)\ s\ \&\ (\%s.\ (\%s.\ True)\ s\ \&\ (\%s.\ True)\ s)\ s$

2 Program Verification

First, we check the program's wellformedness.

lemma *wf-prog*:
wf prog
apply (*simp add*: *wf-def checkPos.simps prog-def Let-def split-def fst-conv snd-conv cmd.simps domC.simps ret-succs.simps callpoints-def isCall-def anF.simps*)
done

Then we prove the verification condition. One simplifier call with definitions of functions used in vc suffices.

lemma *vc-prog-holds*: *prog* $\vdash$ *vc*
apply (*simp add*: *vc-def provable-def valid-def initF-def valid-def Let-def split-def fst-conv snd-conv update-def fun-upd-apply MAX-def lift-def x-def*)
— nprfsize() = 13.136
done

end