theory *EX-DoubleAddition-deep* = *VCOpt*:

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*),
(*JMPB 0*, *Some* (*V x* ≐ *C* (*NAT 12*)))])]

1.3 Verification Condition

We generate an executable ML program for the VCG

generate-code [*term-of*]
  vcg = *vcgSALDeep*
  vcopt = *vcopt*
  prg = *prog*

ML {∗ *set show-brackets*; ∗}
ML {∗ *val vc* = *vcg prog*; ∗}
ML {∗ *val pvc* = (*Pretty.str-of* (*Sign.pretty-term* (*sign-of* (*the-context* ())) (*term-of-form* *vc*))); ∗}
ML {∗ *reset show-brackets* ∗}

constdefs *vc*::*SALform*
vc ≡ Λ [Λ [*Pc* ≐ *C* (*POS* (*0*, *0*)), ∏ *0*. *Deref* (*Lv 0*) ≐ *C ILLEGAL*, ∏ *0*. *Old* (*Deref* (*Lv 0*)) ≐ *C ILLEGAL*, *Rp* ≐ *C* (*POS* (*0*, *0*)), *Tm* ≐ *C* (*NAT 0*)] ⊃ Λ [*C* (*NAT* (*Suc* (*Suc* (*Suc 0*)))) ⪯ *C* (*NAT* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*)))))))))))))))), (*Pc* ≐ *C* (*POS* (*0*, *0*))) ⊃ Λ [(*C* (*NAT* (*Suc* (*Suc* (*Suc 0*)))) ⊕ *C* (*NAT* (*Suc* (*Suc* (*Suc 0*))))) ⪯ *C* (*NAT* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*

0))))))))))))))))), (*C* (*POS* (*0*, *Suc 0*)) ≐ *C* (*POS* (*0*, *Suc 0*))) ⊃ Λ [((*C* (*NAT* (*Suc* (*Suc* (*Suc 0*)))) ⊕ *C* (*NAT* (*Suc* (*Suc* (*Suc 0*))))) ⊕ *C* (*NAT* (*Suc* (*Suc* (*Suc* *0*)))) ⊕ *C* (*NAT* (*Suc* (*Suc* (*Suc 0*))))) ⪯ *C* (*NAT* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*))))))))))))))))), (*C* (*POS* (*0*, *Suc* (*Suc 0*))) ≐ *C* (*POS* (*0*, *Suc* (*Suc 0*)))) ⊃ Λ [*T*, ((*C* (*NAT* (*Suc* (*Suc* (*Suc 0*)))) ⊕ *C* (*NAT* (*Suc* (*Suc* (*Suc 0*))))) ⊕ *C* (*NAT* (*Suc* (*Suc* (*Suc 0*)))) ⊕ *C* (*NAT* (*Suc* (*Suc* (*Suc 0*))))) ≐ *C* (*NAT* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*)))))))))))))]]]], Λ [Λ [Λ [*T*, *V* (*Suc 0*) ≐ *C* (*NAT* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*))))))))))))], *Pc* ≐ *C* (*POS* (*0*, *Suc* (*Suc* (*Suc 0*))))] ⊃ Λ [*T*, *V* (*Suc 0*) ≐ *C* (*NAT* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*))))))))))))]]]

The vc optimiser reduces this formula to T, so the proof becomes trivial

ML {∗ *val vco* = *vcopt* [] *vc*; ∗}

1.4 Program Verification

In this section we prove the vc (without optimisation)

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* ⊢ *vc*
apply (*unfold provable-def*)
apply (*rule HOLprf*)
apply (*simp add*: *valid-def vc-def Let-def split-def fst-conv snd-conv id-lookup-def lift-def nv-def*)
— nprfsize() = 4553
done

In the proof above we instantly shift to HOL by using the rule HOLprf. Instead we can also do large parts of the proof inside our proof calculus and shift to HOL eventually. This technique results in smaller proof objects, because we make less use of the simplifier. The simplifier produces extra load due to rewrites that are not necessary. For example it also simplifies assumptions that are not required for a proof.

lemma *vc-prog-holds-man*: *prog* ⊢ *vc*
apply (*unfold provable-def*)
apply (*unfold vc-def*)
apply (*rule AndI*)
apply (*rule AndSingle*)
apply (*rule AndSingle*)
apply (*rule ImpI*)

```
apply (rule FlattenAsm)
apply (rule Asm)
apply simp
apply (rule ImpI)
apply (rule AndI)
apply (rule AndSingle)
apply (rule FlattenAsm)
apply (rule ImpI)
apply (rule AndI)
apply (rule AndSingle)
apply (rule ImpI)
apply (rule AndI)
apply (rule AndSingle)
apply (rule ImpI)
apply (rule AndI)
apply (rule AndSingle)
apply (rule HOLprf)
apply (rule ballI)
apply (simp (no-asm) add: valid-def validF-validFs.simps lift-def)
apply (rule HOLprf)
apply (rule ballI)
apply (simp (no-asm) add: valid-def validF-validFs.simps lift-def)
apply (rule HOLprf)
apply (rule ballI)
apply (simp (no-asm) add: valid-def validF-validFs.simps lift-def nv-def)
apply (rule HOLprf)
apply (rule ballI)
apply (simp (no-asm) add: valid-def validF-validFs.simps lift-def nv-def)
apply (rule HOLprf)
apply (rule ballI)
apply (simp (no-asm) add: valid-def validF-validFs.simps lift-def nv-def)

done

lemma vc-prog-holds-explict-rules: prog ⊢ vc
apply (unfold vc-def provable-def)
apply (safe intro!: AndI ImpI And0 FlattenAsm FlattenAsmSingle| simp only:
append-Cons append-Nil)+
apply (rule HOLprf, simp add: valid-def vc-def Let-def split-def fst-conv snd-conv
id-lookup-def lift-def nv-def)+

done

end
```