theory *EX-Mult* = *SALTimeFWInst*:

1 Time Bounded Multiplication

We show that a machine program that muliplies two numbers A and B is type safe, contains no overflows and does not require more than MAXTIME computation steps provided the input for A and B is less than MXA or MXB respecitively.

1.1 Variables

We use A and B to store the input values

constdefs
A :: *nat* — Faktor1
A ≡ *0*
B :: *nat* — Faktor2
B ≡ *1*

We use C as counter and R as result variable

constdefs
C::*nat* — Counter
C ≡ *2*
R :: *nat* — Result
R ≡ *3*

Both inputs should not be greater than 3.

constdefs
MXA::*nat*
MXA ≡ *3*
MXB::*nat*
MXB ≡ *3*

We start the program with these values. They could be changed to other values within [0,MXA] or [0,MXB] without affecting the safety proof

constdefs
a0::*nat*
a0 ≡ *3*
b0::*nat*
b0 ≡ *3*

constdefs
prog :: *SALprogram*
prog ≡
[
(*0*,[(*SET R 0*, *None*),
(*SET C 0*, *None*),

(*SET A a0*,*None*),
(*SET B b0*,*None*),
(*JMPEQ C A 4*, *Some* (λ (*p*,*m*,*e*). ∃ *a b r c*. (*m A*)=*NAT a* ∧ (*m B*) =*NAT b* ∧ (*m C*)=*NAT c* ∧ (*m R*)=*NAT r*
∧ *a* <= *MXA* ∧ *b* <= *MXB* ∧ *c* <= *a* ∧ (*r* = *c* ∗ *b*) ∧ (*tim e*) = *4* + *c*∗*4*)),
(*INC C*, *None*),
(*ADD R B*,*None*),
(*JMPB 3*,*None*),
(*HALT*, *None*)
]
)
]

lemma [*code*]:
prog =
[
(*0*,[(*SET R 0*, *None*),
(*SET C 0*, *None*),
(*SET A a0*, *None*),
(*SET B b0*, *None*),
(*JMPEQ C A 4*, *Some* (*term* (λ (*pc*,*m*,*e*). ∃ *a b r c*. (*m A*)=*NAT a* ∧ (*m B*) =*NAT b* ∧ (*m C*)=*NAT c* ∧ (*m R*)=*NAT r*
∧ *a* <= *MXA* ∧ *b* <= *MXB* ∧ *c* <= *a* ∧ (*r* = *c* ∗ *b*) ∧ (*tim e*) = *4* + *c*∗*4*))),
(*INC C*, *None*),
(*ADD R B*,*None*),
(*JMPB 3*,*None*),
(*HALT*, *None*)
]
)
]
apply (*simp only*: *term-def prog-def*)
done

1.2 Verification Condition

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

This is the verification condition the ML program we generated for the VCG above yields.

constdefs
vc::*SALform*
vc ≡ (%*s*. (((%*s*. (((%(*pc*, *m*, *e*). ((*pc* = ((*0*::*nat*), (*0*::*nat*))) & (((*cs e*) = [((*0*::*nat*), *m*)]) & (((*h e*) = []) & (*ALL X*. ((*m X*) = *ILLEGAL*)))))) *s*) −−> ((%*s*. (((%*s*. (((%(*pc*, *m*, *e*). ((*0*::*nat*) <= *MAX*)) *s*) & ((%*s*. (((%(*pc*, *m*, *e*). ((*tim e*) < *MAXTIME*)) *s*) & ((%*s*. *True*) *s*))) *s*))) *s*) & ((%*s*. (((%*s*. (((%(*pc*,

m, e). (pc = ((0::nat), (0::nat)))) s) --> ((%(pc, m, e). ((%s. (((%s. (((%(pc, m, e). ((0::nat) <= MAX)) s) & ((%s. (((%(pc, m, e). ((tim e) < MAXTIME)) s) & ((%s. True) s))) s))) s) & ((%s. (((%s. (((%(pc, m, e). (pc = ((0::nat), (Suc (0::nat))))) s) --> ((%(pc, m, e). ((%s. (((%s. (((%(pc, m, e). ((Suc (Suc (0::nat))) <= MAX)) s) & ((%s. (((%(pc, m, e). ((tim e) < MAXTIME)) s) & ((%s. True) s))) s))) s) & ((%s. (((%s. (((%(pc, m, e). (pc = ((0::nat), (Suc (Suc (0::nat)))))) s) --> ((%(pc, m, e). ((%s. (((%s. (((%(pc, m, e). ((Suc (Suc (Suc (0::nat)))) <= MAX)) s) & ((%s. (((%(pc, m, e). ((tim e) < MAXTIME)) s) & ((%s. True) s))) s))) s) & ((%s. (((%s. (((%(pc, m, e).
(pc = ((0::nat), (Suc (Suc (Suc (0::nat))))))) s) --> ((%(pc, m, e). ((%s. (((%s. (((%(pc, m, e). ((EX n. ((m (Suc (Suc (0::nat)))) = (NAT n))) & (EX n. ((m (0::nat)) = (NAT n))))) s) & ((%s. (((%(pc, m, e). ((tim e) < MAXTIME)) s) & ((%s. True) s))) s))) s) & ((%s. (((%(pc, m, e). (EX a b r c. (((m A) = (NAT a)) & (((m B) = (NAT b)) & (((m C) = (NAT c)) & (((m R) = (NAT r)) & ((a <= MXA) & ((b <= MXB) & ((c <= a) & ((r = (c * b)) & ((tim e) = ((4::nat) + (c * (4::nat))))))))))))) s) & ((%s. True) s))) s))) (((0::nat), (Suc (Suc (Suc (Suc (0::nat)))))), (update m (Suc (0::nat))
(NAT (Suc (Suc (Suc (0::nat))))), (h-update ((h e) @ [((0::nat), (Suc (Suc (Suc (0::nat)))))]) e)))) s))) s) & ((%s. True) s))) s))) (((0::nat), (Suc (Suc (Suc (0::nat))))), (update m (0::nat) (NAT (Suc (Suc (0::nat))))), (h-update ((h e) @ [((0::nat), (Suc (Suc (0::nat))))]) e)))) s))) s) & ((%s. True) s))) s))) (((0::nat), (Suc (Suc (0::nat)))), (update m (Suc (Suc (0::nat))) (NAT (0::nat))), (h-update ((h e) @ [((0::nat), (Suc (0::nat)))]) e)))) s))) s) & ((%s. True) s))) s))) (((0::nat), (Suc (0::nat))), (update m (Suc (Suc (Suc (0::nat)))) (NAT (0::nat))), (h-update ((h e) @ [((0::nat), (0::nat))]) e)))) s))) s) & ((%s. True) s))) s))) s))) s) & ((%s. (((%s. (((%s. (((%s. (((%s. (((%s. (((%(pc, m, e). ((EX n. ((m (Suc (Suc (0::nat)))) = (NAT n))) & (EX n. ((m (0::nat)) = (NAT n))))) s) &
((%s. (((%(pc, m, e). ((tim e) < MAXTIME)) s) & ((%s. True) s))) s))) s) & ((%s. (((%(pc, m, e). (EX a b r c. (((m A) = (NAT a)) & (((m B) = (NAT b)) & (((m C) = (NAT c)) & (((m R) = (NAT r)) & ((a <= MXA) & ((b <= MXB) & ((c <= a) & ((r = (c * b)) & ((tim e) = ((4::nat) + (c * (4::nat))))))))))))) s) & ((%s. True) s))) s))) s) & ((%s. (((%(pc, m, e). ((EX n n'. (((m (Suc (Suc (0::nat)))) = (NAT n)) & (((m (0::nat)) = (NAT n')) & (n = n')))) & (pc = ((0::nat), (Suc (Suc (Suc (Suc (0::nat))))))))) s) & ((%s. True) s))) s))) s) --> ((%(pc, m, e). ((%s. (((%s. (((%s. True) s) & ((%s. (((%(pc, m, e). ((tim e) < MAXTIME)) s) & ((%s. True) s))) s))) s) & ((%s. True) s))) (((0::nat), (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))))), m, (h-update ((h e) @ [((0::nat), (Suc (Suc (Suc (Suc (0::nat))))))]) e)))) s))) s) & ((%s. (((%s. (((%s. (((%s. (((%s. (((%(pc, m, e). ((EX n. ((m (Suc (Suc (0::nat)))) = (NAT n))) & (EX n. ((m (0::nat)) = (NAT n))))) s) &
((%s. (((%(pc, m, e). ((tim e) < MAXTIME)) s) & ((%s. True) s))) s))) s) & ((%s. (((%(pc, m, e). (EX a b r c. (((m A) = (NAT a)) & (((m B) = (NAT b)) & (((m C) = (NAT c)) & (((m R) = (NAT r)) & ((a <= MXA) & ((b <= MXB) & ((c <= a) & ((r = (c * b)) & ((tim e) = ((4::nat) + (c * (4::nat))))))))))))) s) & ((%s. True) s))) s))) s) & ((%s. (((%(pc, m, e). ((EX n n'. (((m (Suc (Suc (0::nat)))) = (NAT n)) & (((m (0::nat)) = (NAT n')) & (n ~= n')))) & (pc = ((0::nat), (Suc (Suc (Suc (Suc (0::nat))))))))) s) & ((%s. True) s))) s))) s) --> ((%(pc, m, e). ((%s. (((%s. (((%(pc, m, e). (EX n. (((m (Suc (Suc (0::nat)))) = (NAT n)) & (n < MAX)))) s) & ((%s. (((%(pc, m, e). ((tim e) < MAXTIME))

s) & ((%s. True) s))) s))) s) & ((%s. (((%s. (((%(pc, m, e). (pc = ((0::nat), (Suc (Suc (Suc (Suc (Suc (0::nat)))))))) s) --> ((%(pc, m, e). ((%s. (((%s. (((%(pc, m, e). ((EX n. ((m (Suc (Suc (Suc (0::nat))))) = (NAT n))) & ((EX n. ((m (Suc (0::nat))) = (NAT n))) &
(EX n. (((lift op + (m (Suc (Suc (Suc (0::nat))))) (m (Suc (0::nat)))) = (NAT n)) & (n <= MAX)))))) s) & ((%s. (((%(pc, m, e). ((tim e) < MAXTIME)) s) & ((%s. True) s))) s))) s) & ((%s. (((%s. (((%(pc, m, e). (pc = ((0::nat), (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))))) s) --> ((%(pc, m, e). ((%s. (((%s. (((%s. True) s) & ((%s. (((%(pc, m, e). (((tim e) < MAXTIME) | ((Suc (Suc (Suc (0::nat)))) = (0::nat)))) s) & ((%s. True) s))) s))) s) & ((%s. (((%s. (((%(pc, m, e). (pc = ((0::nat), (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))))) s) --> ((%(pc, m, e). ((%s. (((%s. (((%(pc, m, e). ((EX n. ((m (Suc (Suc (0::nat)))) = (NAT n))) & (EX n. ((m (0::nat)) = (NAT n))))) s) & ((%s. (((%(pc, m, e). ((tim e) < MAXTIME)) s) & ((%s. True) s))) s))) s) & ((%s. (((%(pc, m, e). (EX a b r c. (((m A) = (NAT a)) & (((m B) = (NAT b)) & (((m C) = (NAT c)) & (((m R) = (NAT r)) & ((a <= MXA) & ((b <= MXB) & ((c <= a) & ((r = (c * b)) & ((tim e) = ((4::nat) + (c * (4::nat)))))))))))))) s) &
((%s. True) s))) s))) (((0::nat), (Suc (Suc (Suc (Suc (0::nat)))))), m, (h-update ((h e) @ [((0::nat), (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))))]) e)))) s))) s) & ((%s. True) s))) s))) (((0::nat), (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))), (update m (Suc (Suc (Suc (0::nat)))) (lift op + (m (Suc (Suc (Suc (0::nat))))) (m (Suc (0::nat))))), (h-update ((h e) @ [((0::nat), (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))))]) e)))) s))) s) & ((%s. True) s))) s))) (((0::nat), (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))), (update m (Suc (Suc (0::nat))) (lift op + (m (Suc (Suc (0::nat)))) (NAT (1::nat)))), (h-update ((h e) @ [((0::nat), (Suc (Suc (Suc (Suc (Suc (0::nat)))))))]) e)))) s))) s) & ((%s. True) s))) s))) (((0::nat), (Suc (Suc (Suc (Suc (Suc (0::nat)))))), m, (h-update ((h e) @ [((0::nat), (Suc (Suc (Suc (Suc (0::nat))))))]) e)))) s))) s) & ((%s. True) s))) s))) s) & ((%s. True) s))) s)))

1.3 Program Verification

First, we check that the program is wellformed

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

lemmas *prog-constants* = *MAX-def MAXTIME-def a0-def b0-def MXA-def MXB-def A-def B-def C-def R-def*
lemmas *prog-simps* = *prog-constants tim-def update-def fun-upd-apply Let-def split-def fst-conv snd-conv initF-def valid-def*

Then we proof the vc. Note that the firt part of the proof is a standard prelude, which is the same for every program. In this prelude we instantiate two system invariants sysinv1 and sysinv2. They give us valuable properties that hold for any wellformed program.

The actual proof for the vc is almost entirely by automatic simplification. They only point where human insight is needed, is where we use lemma about monotonicity of multiplication

lemma *mult-mono*:
⟦ $a \leq (b{::}nat)$; $c \leq (d{::}nat)$ ⟧ $\Longrightarrow a * c \leq b * d$
apply (*simp add*: *mult-le-mono*)
done

theorem *vc-prog-holds*:
provable prog vc
apply (*simp add*: *provable-def valid-def*)
apply (*rule allI*)+
apply (*rename-tac pn i m e*)
apply (*simp only*: *vc-def*)
apply (*rule impI*)
apply (*subgoal-tac sysinv* (((*pn*,*i*),*m*,*e*),*prog*))
 prefer *2*
 apply (*cut-tac wf-prog*)
 apply (*cut-tac sysinv-pres*)
 apply (*erule-tac x*=*prog* **in** *allE*)
 apply (*erule-tac x*=((*pn*,*i*),*m*,*e*) **in** *allE*)
 apply (*drule isafeP-mono*)
 apply (*drule mp*, *assumption*)+
 apply *assumption*
apply (*subgoal-tac sysinv2* (((*pn*,*i*),*m*,*e*),*prog*))
 prefer *2*
 apply (*cut-tac wf-prog*)
 apply (*cut-tac sysinv2-pres*)
 apply (*erule-tac x*=*prog* **in** *allE*)
 apply (*erule-tac x*=((*pn*,*i*),*m*,*e*) **in** *allE*)
 apply (*drule isafeP-mono*)
 apply (*drule mp*, *assumption*)+
 apply *assumption*
apply (*subgoal-tac* ∃ *c css*. *cs e* = *c*#*css*)
 prefer *2*
 apply (*rule classical*)
 apply (*subgoal-tac cs e* = [])
  prefer *2*
  apply (*rule classical*)
  apply (*simp only*: *neq-Nil-conv*)
  apply (*simp add*: *sysinv.simps*)
apply (*erule exE*)+

— main proof
apply (*case-tac prog*,((*pn*,*i*),*m*,*e*) ⊨ (*initF prog*))
apply (*simp add*: *prog-simps*)

— case not (initF prg)
apply (*drule isafeP-mono*)
apply (*erule isafeP-elims*)
apply (*simp add*: *prog-simps*)

apply (*simp add*: *prog-simps*)
apply (*rule conjI*)
apply (*rule impI*)+
apply (*erule conjE* | *erule exE*)+
apply *arith*

apply (*rule impI*)
apply (*erule conjE* | *erule exE*)+
apply (*simp add*: *lift-def nat-number*)
apply (*subgoal-tac* $ca * b \leq (Suc\ (Suc\ 0)) * (Suc\ (Suc\ (Suc\ 0)))$)
 prefer *2*
 apply (*rule mult-mono*)
 apply *simp*
 apply *simp*
apply *simp*
done

end