Chapter 1

Library modified_semantics

```
From Stdlib Require Import Program.Equality.

From Stdlib Require Import Lists.List.
Inductive PartialOrder {A: Type} (rel: A → A → Type) : Type:=
| PartialOrderConstructor (rel_refl: ∀ (a: A), rel a a)
    (rel_trans: ∀ (a b c: A), rel a b → rel b c → rel a c)
    (rel_antisym: ∀ (a b: A), a ≠ b → rel a b → rel b a → False).

Inductive Join {A: Type} (rel: A → A → Type) : A → A → A → Type :=
| JoinConstructor
    (pOrderProof: PartialOrder rel)
    (a b join: A)
    (pleft: rel a join)
    (pright : rel b join)
    (pleast: ∀ ub, rel a ub → rel b ub → rel join ub):
  Join rel a b join
.
Inductive EX {A: Type} (P: A → Type) : Type :=
| EX_intro (x: A) : P x → EX P.

Inductive JoinSemilattice {A: Type} (rel: A → A → Type): Type:=
| JoinSemilatticeConstructor (OrdProof: PartialOrder rel)
    (JoinProof: ∀ (a b: A), EX (fun (join: A) ⇒ Join rel a b join)) .

Inductive Var: Type :=
| VarConstructor (n: nat).

Inductive Level: Type :=
| LevelConstructor (n: nat).

Definition level_eq_dec: ∀ (a b: Level), {a = b} + {a ≠ b}.
Proof.
  decide equality; decide equality.
Qed.
```

```
Definition var_eq_dec: ∀ (a b: Var), {a = b} + {a ≠ b}.
Proof.
  decide equality; decide equality.
Qed.

Inductive BinOp:= | Plus | Minus | Add | Divide | And | Or.

Definition total_map (A: Type) := Var → A.

Definition t_empty {A: Type} (v: A) : total_map A := (fun _ ⇒ v).

Definition t_update {A: Type} (m: total_map A) (x: Var) (v: A) := fun x' ⇒ if var_eq_dec
x x' then v else m x'.

Inductive Primitive :=
| TruePrimitive
| FalsePrimitive
| NatPrimitive (n: nat).

Definition prim_eq_dec: ∀ (a b: Primitive), {a=b} + { a ≠ b}.
Proof.
  repeat (decide equality).
Qed.

Definition MemStore := Var → Primitive × Level.
Definition MemUpdate (mu: MemStore) (x: Var) (p: Primitive) (k: Level) := t_update
mu x (p , k).

Inductive Expression :=
| PrimitiveExpression (prim: Primitive) (k: Level)
| VarExpression (x: Var)
| BinOpExpression (binop: BinOp) (e1 e2: Expression).

Inductive Command : Type :=
| SkipCommand
| AssnCommand (x: Var) (e: Expression)
| SeqCommand (c1 c2: Command)
| IfCommand (e: Expression) (c1 c2: Command)
| WhileCommand (e: Expression) (c: Command)

with DebranchCommand : Type:=
| Debranch (c: Command) (n: bool) (l: Level).

Definition PrimToBool (n: Primitive) : bool := match n with | TruePrimitive ⇒ true | _
⇒ false end.

Inductive Timing :=
| CONST
| VAR
| OPER (oper: BinOp)
```

```
| COMMA
| SKIP
| SEQ
| WHILEF
| ASSN
| IF_HIGH
| IF_LOW
| WHILET

| DEB_SKIP
| DEB_ASSN
| DEB_SEQ
| DEB_IF_HIGH
| DEB_IF_LOW
| DEB_WHILET
| DEB_WHILEF.
Definition timing_eq_dec: ∀ (a b: Timing), {a=b} + {a≠b}.
Proof.
  decide equality; decide equality.
Qed.

Definition TimingList := list Timing.

Definition SingleTiming (t: Timing): TimingList :=
  cons t nil .
Definition AddTiming (t1 t2: TimingList): TimingList := t1 ++ t2.

Notation "a +++ b" := (AddTiming a b) (at level 65, left associativity).

Inductive ExpressionBigStep {binop_eval: BinOp → Primitive → Primitive → Primitive} {rel: Level → Level → Type} {latticeProof: JoinSemilattice rel}: Expression → MemStore → Level → TimingList → Primitive → Level → Type :=
| ConstBigStep (prim: Primitive) {pc k j: Level} (joinProof: Join rel pc k j) (mu: MemStore): ExpressionBigStep (PrimitiveExpression prim k) mu pc (SingleTiming CONST) prim j

| VarBigStep(x: Var) (mu: MemStore) (pc j: Level) (joinProof: Join rel pc (snd (mu x)) j): ExpressionBigStep (VarExpression x) mu pc (SingleTiming VAR) (fst (mu x)) j

| OperBigStep (oper: BinOp) {mu: MemStore}{e1 e2: Expression} {pc k1 k2 joink1k2: Level} {T1 T2: TimingList} {n1 n2: Primitive} (p1: ExpressionBigStep e1 mu pc T1 n1 k1) (p2: ExpressionBigStep e2 mu pc T2 n2 k2) (joinProof: Join rel k1 k2 joink1k2): ExpressionBigStep (BinOpExpression oper e1 e2) mu pc (T1 +++ T2 +++ (SingleTiming (OPER oper))) (binop_eval oper n1 n2) joink1k2.

Inductive CommandBigStep {binop_eval: BinOp → Primitive → Primitive → Prim-
```

itive} {*rel*: **Level** → **Level** → Type} {*latticeProof*: **JoinSemilattice** *rel*}: **Command** → MemStore → **Level** → TimingList → MemStore → Type :=
| SkipBigStep (*mu*: MemStore) (*pc*: **Level**)
  : **CommandBigStep** SkipCommand *mu pc* (SingleTiming SKIP) *mu*

| SeqBigStep {*c1 c2*: **Command**} {*mu mu' mu''*: MemStore} {*pc*: **Level**} {*T1 T2*: TimingList}
    (*p1*: **CommandBigStep** *c1 mu pc T1 mu'*)
    (*p2*: **CommandBigStep** *c2 mu' pc T2 mu''*)
  : **CommandBigStep** (SeqCommand *c1 c2*) *mu pc* (*T1* +++ *T2* +++ (SingleTiming SEQ)) *mu''*

| WhileFBigStep {*e*: **Expression**} {*mu*: MemStore} {*pc k*: **Level**} {*T*: TimingList} (*c*: **Command**) {*n*: **Primitive**}
    (*expressionEvalProof*: (@**ExpressionBigStep** *binop_eval rel latticeProof e mu pc T n k*))
    (*falseProof*: *n* ≠ TruePrimitive)
  : **CommandBigStep** (WhileCommand *e c*) *mu pc* (SingleTiming WHILEF +++ *T*) *mu*

| WhileTBigStep {*e*: **Expression**} {*mu mu' mu''*: MemStore} {*pc k*: **Level**} {*T1 T2 T3*: TimingList} {*c*: **Command**}
    (*expressionEvalProof*: (@**ExpressionBigStep** *binop_eval rel latticeProof e mu pc T1* TruePrimitive *k*))
    (*commandProof*: **CommandBigStep** *c mu pc T2 mu'*)
    (*restLoopProof*: **CommandBigStep** (WhileCommand *e c*) *mu' pc T3 mu''*)
    (*lowProof*: *rel k pc*)
  : **CommandBigStep** (WhileCommand *e c*) *mu pc* (SingleTiming WHILET +++ *T1* +++ *T2* +++ *T3*) *mu''*

| AssnBigStepEq {*e*: **Expression**} {*mu*: MemStore} {*x*: **Var**} {*pc k*: **Level**} {*T*: TimingList} {*n*: **Primitive**}
    (*eproof*: @**ExpressionBigStep** *binop_eval rel latticeProof e mu pc T n k*)
  : **CommandBigStep** (AssnCommand *x e*) *mu pc* (SingleTiming ASSN +++ *T*) (MemUpdate *mu x n k*)

| IfHighBigStep {*e*: **Expression**} {*mu mu' mu''*: MemStore} {*pc kpc*: **Level**} {*n*: **Primitive**} {*T1 T2 T3*: TimingList} {*n*: **Primitive**} {*c1 c2*: **Command**}
    (*eProof*: @**ExpressionBigStep** *binop_eval rel latticeProof e mu pc T1 n kpc*)
    (*debProof1*: **DebranchBigStep** (Debranch *c1* (PrimToBool *n*) *pc*) *mu kpc T2 mu'*)
    (*debProof2*: **DebranchBigStep** (Debranch *c2* (negb (PrimToBool *n*)) *pc*) *mu' kpc T3 mu''*)
    (*relProof*: *rel kpc pc* → **False**)

: **CommandBigStep** (IfCommand e $c1$ $c2$) mu pc (SingleTiming IF_HIGH +++ $T1$ +++ $T2$ +++ $T3$) mu''

| IfLowBigStep
{e: **Expression**} {mu mu': MemStore} {pc k: **Level**} {n: **Primitive**} {$T1$ $T2$: TimingList} ($c1$ $c2$: **Command**)
($eProof$: @**ExpressionBigStep** $binop\_eval$ rel $latticeProof$ e mu pc $T1$ n k)
($relProof$: rel k pc)
($commandProof$: `let` c := `match` n `with` | TruePrimitive ⇒ $c1$ | _ ⇒ $c2$ `end in`
CommandBigStep c mu pc $T2$ mu')
: **CommandBigStep** (IfCommand e $c1$ $c2$) mu pc (SingleTiming IF_LOW +++ $T1$ +++ $T2$) mu'
`with` **DebranchBigStep** {$binop\_eval$: $BinOp \to Primitive \to Primitive \to Primitive$} {$rel$: $Level \to Level \to$ `Type`} {$latticeProof$: $JoinSemilattice$ rel}: **DebranchCommand** → MemStore → **Level** → TimingList → MemStore → `Type` :=

| DebSkipBigStep
(n: bool) (l pc: **Level**) (mu: MemStore)
: **DebranchBigStep** (Debranch SkipCommand n l) mu pc (SingleTiming DEB_SKIP) mu

| DebAssnTrueBigStep {e: **Expression**} {l pc k: **Level**} {mu mu': MemStore} {T: TimingList} {n: **Primitive**}
(x: **Var**)
($evalProof$: @**ExpressionBigStep** $binop\_eval$ rel $latticeProof$ e mu pc T n k)
: **DebranchBigStep** (Debranch (AssnCommand x e) true l) mu pc (SingleTiming DEB_ASSN +++ T) (MemUpdate mu x n k)

| DebAssnFalseBigStep {e: **Expression**} {l pc k: **Level**} {mu: MemStore} {T: TimingList} {n: **Primitive**}
(x: **Var**)
($evalProof$: @**ExpressionBigStep** $binop\_eval$ rel $latticeProof$ e mu pc T n k)
: **DebranchBigStep** (Debranch (AssnCommand x e) false l) mu pc (SingleTiming DEB_ASSN +++ T) (MemUpdate mu x (fst (mu x)) k)

| DebSeqBigStep {$c1$ $c2$: **Command**} {n: bool} {l: **Level**} {mu mu' mu'': MemStore} {l pc: **Level**} {$T1$ $T2$: TimingList}
($p1$: **DebranchBigStep** (Debranch $c1$ n l) mu pc $T1$ mu')
($p2$: **DebranchBigStep** (Debranch $c2$ n l) mu' pc $T2$ mu'')
: **DebranchBigStep** (Debranch (SeqCommand $c1$ $c2$) n l) mu pc (SingleTiming DEB_SEQ +++ $T1$ +++ $T2$) mu''

| DebIfHighBigStep {e: **Expression**} {$c1$ $c2$: **Command**} {mu mu' mu'': MemStore} {l pc

kl kpc: **Level**} {*n*: **bool**} {*n'*: **Primitive**} {*T1 T2 T3 T4*: TimingList}
(*p1*: @**ExpressionBigStep** *binop_eval rel latticeProof e mu l T1 n' kl*)
(*p2*: @**ExpressionBigStep** *binop_eval rel latticeProof e mu pc T2 n' kpc*)
(*relProof*: *rel kl l* → **False**)
(*p3*: **DebranchBigStep** (Debranch *c1* (andb (PrimToBool *n'*) *n*) *l*) *mu kpc T3 mu'*)
(*p4*: **DebranchBigStep** (Debranch *c2* (andb (negb (PrimToBool *n'*)) *n*) *l*) *mu' kpc T4 mu''*)
: **DebranchBigStep** (Debranch (IfCommand *e c1 c2*) *n l*) *mu pc* (SingleTiming DEB_IF_HIGH +++ *T1* +++ *T2* +++ *T3* +++ *T4*) *mu''*
| DebIfLowBigStep
{*e*: **Expression**} {*mu mu'*: MemStore} {*pc k l*: **Level**} {*n*: **bool**} {*n'*: **Primitive**} {*T1 T2*: TimingList} (*c1 c2*: **Command**)
(*eProof*: @**ExpressionBigStep** *binop_eval rel latticeProof e mu l T1 n' k*)

(*relProof*: *rel k l*)
(*commandProof*: `let` *d* := `match` *n'* `with` | TruePrimitive ⇒ (Debranch *c1 n l*) | _ ⇒ (Debranch *c2 n l*) `end in`
DebranchBigStep *d mu pc T2 mu'*)
: **DebranchBigStep** (Debranch (IfCommand *e c1 c2*) *n l*) *mu pc* (SingleTiming DEB_IF_LOW +++ *T1* +++ *T2*) *mu'*
| DebWhileFBigStep {*e*: **Expression**} {*mu*: MemStore} {*k l*: **Level**} {*T*: TimingList} {*n'*: **Primitive**}
(*c*: **Command**)
(*n*: **bool**)
(*pc*: **Level**)
(*expressionEvalProof*: (@**ExpressionBigStep** *binop_eval rel latticeProof e mu l T n' k*))
(*falseProof*: *n'* ≠ TruePrimitive)
: **DebranchBigStep** (Debranch (WhileCommand *e c*) *n l*) *mu pc* (SingleTiming DEB_WHILEF +++ *T*) *mu*

| DebWhileTBigStep {*e*: **Expression**} {*mu mu' mu''*: MemStore} {*pc l kl kpc*: **Level**} {*T1 T1' T2 T3*: TimingList} {*c*: **Command**} {*n*: **bool**}
(*expressionEvalProof*: (@**ExpressionBigStep** *binop_eval rel latticeProof e mu l T1* TruePrimitive *kl*))
(*commandProof*: **DebranchBigStep** (Debranch *c n l*) *mu pc T2 mu'*)
(*restLoopProof*: **DebranchBigStep** (Debranch (WhileCommand *e c*) *n l*) *mu' pc T3 mu''*)
(*lowProof*: *rel kl l*)
: **DebranchBigStep** (Debranch (WhileCommand *e c*) *n l*) *mu pc* (SingleTiming DEB_WHILET +++ *T1* +++ *T2* +++ *T3*) *mu''*.

`Inductive` **ValueObservationalEquivalent** {*rel*: **Level** → **Level** → Type} {*latticeProof*:

JoinSemilattice *rel*}: **Primitive** → **Level** → **Level** → **Primitive** → **Level** → Type :=
| LowProof {*n1 n2*: **Primitive**} {*l1 l2 l*: **Level**} (*nEq*: *n1* = *n2*) (*lEq*: *l1* = *l2*) : **ValueObservationalEquivalent** *n1 l1 l n2 l2*
| HighProof (*n1 n2*: **Primitive**) {*l1 l2 l*: **Level**} (*l1High*: *rel l1 l* → **False**) (*l2High*: *rel l2 l* → **False**): **ValueObservationalEquivalent** *n1 l1 l n2 l2*.

`Definition` MemStoreObservationalEquivalent {*rel*: **Level** → **Level** → Type} {*latticeProof*: **JoinSemilattice** *rel*} (*mu1*: MemStore) (*l*: **Level**) (*mu2*: MemStore): Type := ∀ (*x*: **Var**), @**ValueObservationalEquivalent** *rel latticeProof* (fst (*mu1 x*)) (snd (*mu1 x*)) *l* (fst (*mu2 x*)) (snd (*mu2 x*)).

`Lemma` MemStoreObservationalEquivalentRefl {*rel*: **Level** → **Level** → Type} {*latticeProof*: **JoinSemilattice** *rel*} (*mu1*: MemStore) (*l*: **Level**) (*mu2*: MemStore):
  ∀ *mu pc*,
    @MemStoreObservationalEquivalent *rel latticeProof mu pc mu*.
`Proof.`
  `intros` *mu pc*. `intros` *x*. `apply` LowProof; `reflexivity`.
`Qed.`

`Lemma` JoinEq: ∀ {*rel*: **Level** → **Level** → Type} (*latticeProof*: **JoinSemilattice** *rel*) {*a b j1 j2*: **Level**}, **Join** *rel a b j1* → **Join** *rel a b j2* → *j1* = *j2*.
`Proof.`
  `intros.` `destruct` *X*; `destruct` *X0*; `destruct` *latticeProof*; `destruct` *OrdProof*. `destruct` (level_eq_dec *join join0*).
  - `assumption.`
  - `specialize` (*pleast join0 pleft0 pright0*); `specialize` (*pleast0 join pleft pright*). `specialize` (*rel_antisym join join0 n pleast pleast0*). *contradiction*.
`Qed.`
`Lemma` JoinSym: ∀ {*rel*: **Level** → **Level** → Type} (*latticeProof*: **JoinSemilattice** *rel*) {*a b join*: **Level**}, **Join** *rel a b join* → **Join** *rel b a join*.
`Proof.`
  `intros.` `destruct` *X*. `constructor`; (`try` `assumption`).
  - `intros.` `apply` (*pleast ub X0 X*).
`Qed.`

`Lemma` JoinHigh: ∀ {*rel*: **Level** → **Level** → Type} (*latticeProof*: **JoinSemilattice** *rel*) {*H L X joinHX*: **Level**}, (**Join** *rel H X joinHX*) → (*rel H L* → **False**) → (*rel joinHX L* → **False**).
`Proof.`
  `intros.` `destruct` *X0*; `destruct` *latticeProof*; `destruct` *OrdProof*. `apply` *H0*. `apply` (*rel_trans _ _ _ pleft X1*).
`Qed.`

`Lemma` RelAlwaysRefl {*rel*: **Level** → **Level** → Type} (*latticeProof*: **JoinSemilattice** *rel*) {*l*: **Level**}: (*rel l l* → **False**) → **False**.
`Proof.`

intros. destruct *latticeProof*; destruct *OrdProof*. apply (*H* (*rel_refl l*)).
Qed.
Lemma NotRelImplNotEq: ∀ {*rel*: **Level** → **Level** → Type} (*latticeProof*: **JoinSemilattice** *rel*) {*l1 l2*: **Level**}, (*rel l1 l2* → **False**) → $l1 \neq l2$.
Proof.
intros. destruct (level_eq_dec *l1 l2*).
- subst. destruct *latticeProof*; destruct *OrdProof*. specialize (*H* (*rel_refl l2*)). *contradiction*.
- assumption.
Qed.

Lemma ExpressionLabelLowerBound: ∀ {*binop_eval*: **BinOp** → **Primitive** → **Primitive** → **Primitive**} {*rel*: **Level** → **Level** → Type} {*latticeProof*: **JoinSemilattice** *rel*} {*e*: **Expression**} {*mu*: MemStore} {*l k*: **Level**} {*T*: TimingList} {*n*: **Primitive**} (*proof*: @**ExpressionBigStep** *binop_eval rel latticeProof e mu l T n k*), *rel l k*.
Proof.
intros. induction *proof*.
- destruct *joinProof*. assumption.
- destruct *joinProof*. assumption.
- destruct *latticeProof*. destruct *OrdProof*. destruct *joinProof*. apply (*rel_trans* _ _ _ *IHproof1 pleft*).

Qed.

Lemma ExpressionLabelLowestBound: ∀ {*binop_eval*: **BinOp** → **Primitive** → **Primitive** → **Primitive**} {*rel*: **Level** → **Level** → Type} {*latticeProof*: **JoinSemilattice** *rel*} {*e*: **Expression**} {*mu*: MemStore} {*l k*: **Level**} {*T*: TimingList} {*n*: **Primitive**} (*proof*: @**ExpressionBigStep** *binop_eval rel latticeProof e mu l T n k*), *rel k l* → *l* = *k*.
Proof.
intros. pose *proof* (ExpressionLabelLowerBound *proof*). destruct (level_eq_dec *l k*).
- assumption.
- destruct *latticeProof*; destruct *OrdProof*. pose *proof* (*rel_antisym* _ _ *n0 X0 X*). *contradiction*.
Qed.

Lemma BiggerFish {*rel*: **Level** → **Level** → Type} {*latticeProof*: **JoinSemilattice** *rel*}: ∀ {*LL L H*: **Level**},
$LL \neq L$ → *rel LL L* → *rel L H* → (*rel H LL* → **False**).
Proof.
intros. destruct *latticeProof*; destruct *OrdProof*. destruct (level_eq_dec *L H*).
- subst. apply (*rel_antisym* _ _ *H0 X X1*).
- specialize (*rel_trans* _ _ _ *X0 X1*). apply (*rel_antisym* _ _ *H0 X rel_trans*).
Qed.

Lemma MemStoreEquivalenceImplExpressionEquivalence:
∀ {*binop_eval*: **BinOp** → **Primitive** → **Primitive** → **Primitive**} {*rel*: **Level** → **Level** →

```
Type} {latticeProof: JoinSemilattice rel} {e: Expression} {mu1 mu2: MemStore} {l k1
k2: Level} {n1 n2: Primitive} {T1 T2: TimingList}
          (p1: @ExpressionBigStep binop_eval rel latticeProof e mu1 l T1 n1 k1)
          (p2: @ExpressionBigStep binop_eval rel latticeProof e mu2 l T2 n2 k2)
          (memEqProof: @MemStoreObservationalEquivalent rel latticeProof mu1 l mu2),
    @ValueObservationalEquivalent rel latticeProof n1 k1 l n2 k2.
Proof.
  intros.
  dependent induction e; dependent destruction p1; dependent destruction p2.
  -
    pose proof (JoinEq latticeProof joinProof0 joinProof).
    subst j. constructor; auto.
  - specialize (memEqProof x). destruct memEqProof.
    + subst. pose proof (JoinEq latticeProof joinProof joinProof0). subst. constructor;
auto.
    + specialize (JoinHigh latticeProof (JoinSym latticeProof joinProof) l1High);
        specialize (JoinHigh latticeProof (JoinSym latticeProof joinProof0) l2High); intros.
apply (HighProof n1 n2 H0 H).
  - specialize (IHe1 _ _ _ _ _ _ _ _ _ _ p1_1 p2_1 memEqProof); specialize (IHe2 _ _
_ _ _ _ _ _ _ _ p1_2 p2_2 memEqProof). destruct IHe1; destruct IHe2.
     + subst. pose proof (JoinEq latticeProof joinProof joinProof0). subst. constructor;
auto.
     + subst. specialize (JoinHigh latticeProof (JoinSym latticeProof joinProof) l1High);
         specialize (JoinHigh latticeProof (JoinSym latticeProof joinProof0) l2High); intros.
apply (HighProof _ _ H0 H).
     + subst. specialize (JoinHigh latticeProof joinProof l1High);
         specialize (JoinHigh latticeProof joinProof0 l2High); intros. apply (HighProof
_ _ H0 H).
     + subst. specialize (JoinHigh latticeProof joinProof l1High);
         specialize (JoinHigh latticeProof joinProof0 l2High); intros. apply (HighProof
_ _ H0 H).
Qed.

Inductive LoopLengthCommand {binop_eval: BinOp → Primitive → Primitive →
Primitive} {rel: Level → Level → Type} {latticeProof: JoinSemilattice rel} : Level
→ MemStore → Expression → Command → TimingList → MemStore → nat → Type :=
| LoopLengthCommand0 {mu: MemStore} {e: Expression} {n: Primitive} {pc k: Level}
{T: TimingList}
    (c: Command)
    (expressionEvalProof: @ExpressionBigStep binop_eval rel latticeProof e mu pc T n
k)
    (primProof: n ≠ TruePrimitive)
```

: **LoopLengthCommand** *pc mu e c* (SingleTiming WHILEF +++ *T*) *mu* 0

| LoopLengthCommandSn {*mu mu' mu''*: MemStore} {*e*: **Expression**} {*n*: **nat**} {*pc k*: **Level**} {*Te Tc Tw*: TimingList} {*c*: **Command**}
(*expressionProof*: @**ExpressionBigStep** *binop_eval rel latticeProof e mu pc Te* TruePrimitive *k*)
(*commandProof*: @**CommandBigStep** *binop_eval rel latticeProof c mu pc Tc mu'*)
(*whileProof*: @**CommandBigStep** *binop_eval rel latticeProof* (WhileCommand *e c*) *mu' pc Tw mu''*)
(*indProof*: **LoopLengthCommand** *pc mu' e c Tw mu'' n*)
(*relProof*: *rel k pc*)
: **LoopLengthCommand** *pc mu e c* (SingleTiming WHILET +++ *Te* +++ *Tc* +++ *Tw*) *mu''* (S *n*).

Lemma AlwaysLoopLengthCommand {*binop_eval*: **BinOp** → **Primitive** → **Primitive** → **Primitive**} {*rel*: **Level** → **Level** → Type} {*latticeProof*: **JoinSemilattice** *rel*}:
∀ {*e*: **Expression**} {*c*: **Command**} {*mu mu'*: MemStore} {*pc*: **Level**} {*T*: TimingList},
@**CommandBigStep** *binop_eval rel latticeProof* (WhileCommand *e c*) *mu pc T mu'* →
EX (fun *n* ⇒ @**LoopLengthCommand** *binop_eval rel latticeProof pc mu e c T mu' n*).

Proof.
intros.
dependent induction *X*.
- apply (EX_intro _ 0 (LoopLengthCommand0 *c expressionEvalProof falseProof*)).
- clear *IHX1*. assert (WhileCommand *e c* = WhileCommand *e c*) by auto. specialize (*IHX2 H*); clear *H*; destruct *IHX2*. apply (EX_intro _ (S *x*) (LoopLengthCommandSn *expressionEvalProof X1 X2 l lowProof*)).
Qed.

Lemma MemStoreEquivalenceImplLoopLengthCommandEq {*binop_eval*: **BinOp** → **Primitive** → **Primitive** → **Primitive**} {*rel*: **Level** → **Level** → Type} {*latticeProof*: **JoinSemilattice** *rel*}:
∀ {*e*: **Expression**} {*c*: **Command**} {*T1 T2*: TimingList} {*mu1 mu1' mu2 mu2'*: MemStore} {*pc*: **Level**} {*n1 n2*: **nat**}
(*cMemEq*: ∀ (*mu1 mu2 mu1' mu2'* : MemStore) (*pc* : **Level**) (*T1 T2* : TimingList),
@**CommandBigStep** *binop_eval rel latticeProof c mu1 pc T1 mu1'* →
@**CommandBigStep** *binop_eval rel latticeProof c mu2 pc T2 mu2'* →
@MemStoreObservationalEquivalent *rel latticeProof mu1 pc mu2* → @MemStoreObservationalEquivalent *rel latticeProof mu1' pc mu2'*),
@**LoopLengthCommand** *binop_eval rel latticeProof pc mu1 e c T1 mu1' n1* →
@**LoopLengthCommand** *binop_eval rel latticeProof pc mu2 e c T2 mu2' n2* →
@MemStoreObservationalEquivalent *rel latticeProof mu1 pc mu2* →
n1 = *n2*.

```
Proof.
  intros. generalize dependent mu2. generalize dependent mu2'. generalize dependent
mu1'. generalize dependent mu1; revert T2; revert T1. dependent induction n1;
intros; dependent destruction X; dependent destruction X0.
    + reflexivity.
    + pose proof (MemStoreEquivalenceImplExpressionEquivalence expressionEvalProof expressionProof X1). destruct H; contradiction.
    + pose proof (MemStoreEquivalenceImplExpressionEquivalence expressionProof expressionEvalProof X1). destruct H; subst; contradiction.
    + pose proof (cMemEq _ _ _ _ _ _ _ _ commandProof commandProof0 X1).
      specialize (IHn1 _ cMemEq _ _ _ _ X _ _ X0 X2).
      subst. reflexivity.
Qed.
```

```
Inductive LoopLengthDebranch {binop_eval: BinOp → Primitive → Primitive → Primitive} {rel: Level → Level → Type} {latticeProof: JoinSemilattice rel} : Level → MemStore → Expression → Command → bool → Level → TimingList → MemStore → nat → Type :=
| LoopLengthDebranch0 {mu: MemStore} {e: Expression} {n': Primitive} {l k: Level} {T: TimingList}
    (c: Command) (n: bool) (pc: Level)
    (expressionEvalProof: @ExpressionBigStep binop_eval rel latticeProof e mu l T n' k)
    (primProof: n' ≠ TruePrimitive)
  : LoopLengthDebranch pc mu e c n l (SingleTiming DEB_WHILEF +++ T) mu 0

| LoopLengthDebranchSn {mu mu' mu'': MemStore} {e: Expression} {x: nat} {pc l kl: Level} {Te Tc Tw: TimingList} {c: Command} {n: bool}
    (expressionEvalProof: (@ExpressionBigStep binop_eval rel latticeProof e mu l Te TruePrimitive kl))
    (commandProof: @DebranchBigStep binop_eval rel latticeProof (Debranch c n l) mu pc Tc mu')
    (restLoopProof: @DebranchBigStep binop_eval rel latticeProof (Debranch (WhileCommand e c) n l) mu' pc Tw mu'')
    (indProof: LoopLengthDebranch pc mu' e c n l Tw mu'' x)
    (lowProof: rel kl l)

  : LoopLengthDebranch pc mu e c n l (SingleTiming DEB_WHILET +++ Te +++ Tc +++ Tw) mu'' (S x).
```

```
Lemma AlwaysLoopLengthDebranch {binop_eval: BinOp → Primitive → Primitive → Primitive} {rel: Level → Level → Type} {latticeProof: JoinSemilattice rel}:
  ∀ {e: Expression} {c: Command} {n: bool} {mu mu': MemStore} {pc l: Level} {T:
```

TimingList},
    @**DebranchBigStep** *binop_eval rel latticeProof* (Debranch (WhileCommand *e c*) *n l*) *mu pc T mu'* →
    EX (fun *x* ⇒ @**LoopLengthDebranch** *binop_eval rel latticeProof pc mu e c n l T mu' x*).

```
Proof.
  intros.
  dependent induction X.
  - apply (EX_intro _ 0 (LoopLengthDebranch0 c n pc expressionEvalProof falseProof)).
  - clear IHX1. assert (Debranch (WhileCommand e c) n l = Debranch (WhileCommand e c) n l) by auto. specialize (IHX2 H); clear H; destruct IHX2.
    apply (EX_intro _ (S x) (LoopLengthDebranchSn expressionEvalProof X1 X2 l0 lowProof)).
Qed.
```

Lemma MemStoreEquivalenceImplLoopLengthDebranchEq {*binop_eval*: **BinOp** → **Primitive** → **Primitive** → **Primitive**} {*rel*: **Level** → **Level** → Type} {*latticeProof*: **JoinSemilattice** *rel*}:
  ∀ {*e*: **Expression**} {*c*: **Command**} {*n1 n2*: **bool**} {*mu1 mu1' mu2 mu2'*: MemStore} {*T1 T2*: TimingList} {*pc1 pc2 l*: **Level**} {*x1 x2*: **nat**}

        (*debcMemEq*: ∀ (*n1 n2* : **bool**) (*mu1 mu2 mu1' mu2'* : MemStore) (*l pc1 pc2* : **Level**) (*T1 T2* : TimingList),
        @**DebranchBigStep** *binop_eval rel latticeProof* (Debranch *c n1 l*) *mu1 pc1 T1 mu1'* →
        @**DebranchBigStep** *binop_eval rel latticeProof*(Debranch *c n2 l*) *mu2 pc2 T2 mu2'* →
        @MemStoreObservationalEquivalent *rel latticeProof mu1 l mu2* →
        rel l pc1 → *l* ≠ *pc1* → *rel l pc2* → *l* ≠ *pc2* → @MemStoreObservationalEquivalent *rel latticeProof mu1' l mu2'*),
    rel l pc1 → *l* ≠ *pc1* → *rel l pc2* → *l* ≠ *pc2* →
    @**LoopLengthDebranch** *binop_eval rel latticeProof pc1 mu1 e c n1 l T1 mu1' x1* →
    @**LoopLengthDebranch** *binop_eval rel latticeProof pc2 mu2 e c n2 l T2 mu2' x2* →
    @MemStoreObservationalEquivalent *rel latticeProof mu1 l mu2* →
    x1 = *x2*.

```
Proof.
  intros. generalize dependent mu2; generalize dependent mu2'; generalize dependent mu1'; generalize dependent mu1; revert T2; revert T1. dependent induction x1; intros; dependent destruction X1; dependent destruction X2.
    + reflexivity.
    + assert (n' = TruePrimitive). {

        pose proof (MemStoreEquivalenceImplExpressionEquivalence expressionEvalProof ex-
```

```
pressionEvalProof0 X3).
          pose proof (ExpressionLabelLowestBound expressionEvalProof0 lowProof); subst kl.
          remember latticeProof; destruct j; destruct OrdProof.
          dependent destruction H1.
          - auto.
          - specialize (l2High (rel_refl l)). contradiction.
        } contradiction.
      + assert (n' = TruePrimitive). {
          pose proof (MemStoreEquivalenceImplExpressionEquivalence expressionEvalProof expressionEvalProof0 X3).
          pose proof (ExpressionLabelLowestBound expressionEvalProof lowProof); subst kl.
          remember latticeProof; destruct j; destruct OrdProof.
          dependent destruction H1.
          - auto.
          - specialize (l1High (rel_refl l1)). contradiction.
         } contradiction.
      +

        pose proof (ExpressionLabelLowestBound expressionEvalProof lowProof); subst kl;
          pose proof (ExpressionLabelLowestBound expressionEvalProof0 lowProof0); subst kl0; clear lowProof lowProof0.

        pose proof (debcMemEq _ _ _ _ _ _ _ _ _ _ _ _ commandProof commandProof0 X3 X H X0 H0).
        specialize (IHx1 _ debcMemEq X H X0 H0 _ _ _ _ X1 _ _ X2 X4).
        subst. reflexivity.
Qed.

Theorem DebranchPreservesMemEq {binop_eval: BinOp → Primitive → Primitive → Primitive} {rel: Level → Level → Type} {latticeProof: JoinSemilattice rel} : ∀ {c: Command} {n1 n2: bool} {mu1 mu2 mu1' mu2': MemStore} {l pc1 pc2: Level} {T1 T2: TimingList}
            (p1: @DebranchBigStep binop_eval rel latticeProof (Debranch c n1 l) mu1 pc1 T1 mu1')
            (p2: @DebranchBigStep binop_eval rel latticeProof (Debranch c n2 l) mu2 pc2 T2 mu2')
            (memEq: @MemStoreObservationalEquivalent rel latticeProof mu1 l mu2)
            (l_rel_pc1: rel l pc1)
            (l_not_pc1: l ≠ pc1)
            (l_rel_pc2: rel l pc2)
            (l_not_pc2: l ≠ pc2),
      @MemStoreObservationalEquivalent rel latticeProof mu1' l mu2'.
Proof.
```

```
intros. dependent induction c.

- dependent destruction p1; dependent destruction p2. assumption.

- dependent destruction p1; dependent destruction p2; unfold MemStoreObservationalEquivalent in *; unfold MemUpdate; unfold t_update; intros; destruct (var_eq_dec x x0); auto; simpl; subst; pose proof (ExpressionLabelLowerBound evalProof); pose proof (ExpressionLabelLowerBound evalProof0); pose proof (@BiggerFish rel latticeProof _ _ _ l_not_pc1 l_rel_pc1 X); pose proof (@BiggerFish rel latticeProof _ _ _ l_not_pc2 l_rel_pc2 X0); apply (HighProof _ _ H H0).

- dependent destruction p1; dependent destruction p2. specialize (IHc1 _ _ _ _ _ _ _ _ _ _ _ _ p1_1 p2_1 memEq l_rel_pc1 l_not_pc1 l_rel_pc2 l_not_pc2). apply (IHc2 _ _ _ _ _ _ _ _ _ _ _ _ p1_2 p2_2 IHc1 l_rel_pc1 l_not_pc1 l_rel_pc2 l_not_pc2).

- dependent destruction p2; dependent destruction p1.
  + pose proof (ExpressionLabelLowerBound p2); pose proof (ExpressionLabelLowerBound p3).

    remember latticeProof; destruct j; destruct OrdProof.

    pose proof (rel_trans _ _ _ l_rel_pc1 X).
    pose proof (rel_trans _ _ _ l_rel_pc2 X0).

    assert (eq: l ≠ kpc ∧ l ≠ kpc0). {
      split.
      - destruct (level_eq_dec l kpc); auto. subst. pose proof (rel_antisym _ _ l_not_pc1 l_rel_pc1 X). contradiction.
      - destruct (level_eq_dec l kpc0); auto. subst. pose proof (rel_antisym _ _ l_not_pc2 l_rel_pc2 X0). contradiction.
    } destruct eq as [eq eq0].

    specialize (IHc1 _ _ _ _ _ _ _ _ _ _ _ _ _ p1_1 p2_1 memEq X1 eq X2 eq0).
    apply (IHc2 _ _ _ _ _ _ _ _ _ _ _ _ _ p1_2 p2_2 IHc1 X1 eq X2 eq0).
  + assert(k = kl). {
      Check @MemStoreEquivalenceImplExpressionEquivalence.
      pose proof (MemStoreEquivalenceImplExpressionEquivalence eProof p0 memEq).
      destruct H.
      - assumption.
      - contradiction.
    } subst. contradiction.

  + assert(k = kl). {

      pose proof (MemStoreEquivalenceImplExpressionEquivalence p1 eProof memEq).
      destruct H.
      - auto.
      - contradiction.
```

```
    } subst. contradiction.
  + assert (n' = n'0 ∧ k0 = k). {

      pose proof (MemStoreEquivalenceImplExpressionEquivalence eProof eProof0 memEq).
      destruct H.
      - split; auto.
      - contradiction.
    } destruct H. subst.
    destruct n'0;
      try (apply (IHc1 _ _ _ _ _ _ _ _ _ _ _ _ commandProof commandProof0 memEq
l_rel_pc1 l_not_pc1 l_rel_pc2 l_not_pc2));
      try (apply (IHc2 _ _ _ _ _ _ _ _ _ _ _ _ commandProof commandProof0 memEq
l_rel_pc1 l_not_pc1 l_rel_pc2 l_not_pc2)).
- pose proof (AlwaysLoopLengthDebranch p1); pose proof (AlwaysLoopLengthDebranch
p2). destruct X; destruct X0. Check @MemStoreEquivalenceImplLoopLengthDebranchEq.
  pose proof (MemStoreEquivalenceImplLoopLengthDebranchEq IHc l_rel_pc1 l_not_pc1
l_rel_pc2 l_not_pc2 l0 l1 memEq); subst x0. clear p2; clear p1. revert memEq. generalize
dependent mu2'. revert mu2. generalize dependent mu1'. revert mu1. revert T2; revert
T1. dependent induction x.
  + intros. dependent destruction l0; dependent destruction l1. assumption.
  + intros. dependent destruction l0; dependent destruction l1.

    pose proof (ExpressionLabelLowestBound expressionEvalProof lowProof); subst kl.
      pose proof (ExpressionLabelLowestBound expressionEvalProof0 lowProof0); subst
kl0; clear lowProof lowProof0.
      specialize (IHc _ _ _ _ _ _ _ _ _ _ _ _ commandProof commandProof0 memEq
l_rel_pc1 l_not_pc1 l_rel_pc2 l_not_pc2).
      apply (IHx _ _ _ _ l0 _ _ l1 IHc).
Qed.

Theorem CommandPreservesMemEq {binop_eval: BinOp → Primitive → Primitive → Prim-
itive} {rel: Level → Level → Type} {latticeProof: JoinSemilattice rel} :
  ∀ {c: Command} {mu1 mu2 mu1' mu2': MemStore} {pc: Level} {T1 T2: TimingList}
    (p1: @CommandBigStep binop_eval rel latticeProof c mu1 pc T1 mu1')
    (p2: @CommandBigStep binop_eval rel latticeProof c mu2 pc T2 mu2')
    (memEq: @MemStoreObservationalEquivalent rel latticeProof mu1 pc mu2),
  @MemStoreObservationalEquivalent rel latticeProof mu1' pc mu2'.
Proof.
  intros. dependent induction c.
  - dependent destruction p1; dependent destruction p2. assumption.
  - dependent destruction p1; dependent destruction p2. unfold MemStoreObserva-
tionalEquivalent in *; unfold MemUpdate; unfold t_update. intros; destruct (var_eq_dec
x x0).
```

```
+ simpl. pose proof (MemStoreEquivalenceImplExpressionEquivalence eproof eproof0
memEq). assumption.
+ specialize (memEq x0). assumption.
- dependent destruction p1; dependent destruction p2.
specialize (IHc1 _ _ _ _ _ _ _ _ p1_1 p2_1 memEq).
apply (IHc2 _ _ _ _ _ _ _ _ p1_2 p2_2 IHc1).
- dependent destruction p1; dependent destruction p2.
+ assert (notRel: pc ≠ kpc) by (apply not_eq_sym; apply (NotRelImplNotEq lattice-
Proof relProof)).
assert (notRel0: pc ≠ kpc0) by (apply not_eq_sym; apply (NotRelImplNotEq latti-
ceProof relProof0)).
assert (low: rel pc kpc) by (apply (ExpressionLabelLowerBound eProof)).
assert (low0: rel pc kpc0) by (apply (ExpressionLabelLowerBound eProof0)).
pose proof (DebranchPreservesMemEq debProof1 debProof0 memEq low notRel low0
notRel0).
apply (DebranchPreservesMemEq debProof2 debProof3 X low notRel low0 notRel0).
+ pose proof (ExpressionLabelLowestBound eProof0 relProof0); subst k.
assert (kpc = pc). {
pose proof (MemStoreEquivalenceImplExpressionEquivalence eProof eProof0 memEq).
dependent destruction H.
- reflexivity.
- pose proof (RelAlwaysRefl latticeProof l2High). contradiction.
} subst. pose proof (RelAlwaysRefl latticeProof relProof). contradiction.
+ pose proof (ExpressionLabelLowestBound eProof relProof); subst k.
assert (kpc = pc). {
pose proof (MemStoreEquivalenceImplExpressionEquivalence eProof eProof0 memEq).
dependent destruction H.
- reflexivity.
- pose proof (RelAlwaysRefl latticeProof l1High). contradiction.
} subst. pose proof (RelAlwaysRefl latticeProof relProof0). contradiction.
+ pose proof (ExpressionLabelLowestBound eProof relProof); subst k. pose proof (Ex-
pressionLabelLowestBound eProof0 relProof0); subst k0.
assert (n=n0). {
pose proof (MemStoreEquivalenceImplExpressionEquivalence eProof eProof0 memEq).
dependent destruction H.
- reflexivity.
- pose proof (RelAlwaysRefl latticeProof l1High). contradiction.
} subst n0. destruct n;
try (apply (IHc1 _ _ _ _ _ _ _ _ commandProof commandProof0 memEq));
try (apply (IHc2 _ _ _ _ _ _ _ _ commandProof commandProof0 memEq)).
- pose proof (AlwaysLoopLengthCommand p1); (pose proof (AlwaysLoopLengthCommand
p2)). destruct X as [n l1]. destruct X0 as [n1 l2]. clear p1; clear p2. pose proof
```

```
(MemStoreEquivalenceImplLoopLengthCommandEq IHc l1 l2 memEq). subst n1. generalize
dependent memEq. generalize dependent mu2'; generalize dependent mu2; generalize
dependent mu1'; generalize dependent mu1; revert T2; revert T1. dependent induction
n; intros.
    + dependent destruction l1; dependent destruction l2. assumption.
    + dependent destruction l1; dependent destruction l2.
      specialize (IHc _ _ _ _ _ _ _ _ commandProof commandProof0 memEq).
      apply (IHn _ _ _ _ l1 _ _ l2 IHc).
Qed.

Lemma ExpressionTimSec {binop_eval: BinOp → Primitive → Primitive → Primitive} {rel:
Level → Level → Type} {latticeProof: JoinSemilattice rel}:
  ∀ {e: Expression} {pc1 pc2 k1 k2: Level} {mu1 mu2: MemStore} {n1 n2: Primitive}
{T1 T2: TimingList},
    @ExpressionBigStep binop_eval rel latticeProof e mu1 pc1 T1 n1 k1 →
    @ExpressionBigStep binop_eval rel latticeProof e mu2 pc2 T2 n2 k2 →
    (T1 = T2).
Proof.
  intros. dependent induction e; dependent destruction X0; dependent destruction
X.
  - reflexivity.
  - reflexivity.
  - specialize (IHe1 _ _ _ _ _ _ _ _ _ _ _ X1 X0_1).
    specialize (IHe2 _ _ _ _ _ _ _ _ _ _ _ X2 X0_2).
    rewrite → IHe1.
    rewrite → IHe2.
    reflexivity.
Qed.

Lemma DebranchTimSec {binop_eval: BinOp → Primitive → Primitive → Primitive} {rel:
Level → Level → Type} {latticeProof: JoinSemilattice rel}:
  ∀ {c: Command} {n1 n2: bool} {mu1 mu2 mu1' mu2': MemStore} {l pc1 pc2: Level}
{T1 T2: TimingList}
          (p1: @DebranchBigStep binop_eval rel latticeProof (Debranch c n1 l) mu1 pc1
T1 mu1')
          (p2: @DebranchBigStep binop_eval rel latticeProof (Debranch c n2 l) mu2 pc2
T2 mu2')
          (memEq: @MemStoreObservationalEquivalent rel latticeProof mu1 l mu2)
          (l_rel_pc1: rel l pc1)
          (l_not_pc1: l ≠ pc1)
          (l_rel_pc2: rel l pc2)
          (l_not_pc2: l ≠ pc2),
    T1 = T2.
Proof.
```

```
intros. dependent induction c.
- dependent destruction p1; dependent destruction p2. reflexivity.
- dependent destruction p1; dependent destruction p2; pose proof (ExpressionTimSec evalProof evalProof0); subst; reflexivity.
- dependent destruction p1; dependent destruction p2.
  pose proof (DebranchPreservesMemEq p1_1 p2_1 memEq l_rel_pc1 l_not_pc1 l_rel_pc2 l_not_pc2) as memEq0.
  specialize (IHc1 _ _ _ _ _ _ _ _ _ _ _ _ p1_1 p2_1 memEq l_rel_pc1 l_not_pc1 l_rel_pc2 l_not_pc2).
  specialize (IHc2 _ _ _ _ _ _ _ _ _ _ _ _ p1_2 p2_2 memEq0 l_rel_pc1 l_not_pc1 l_rel_pc2 l_not_pc2).
  subst. reflexivity.

- dependent destruction p2; dependent destruction p1.
  + pose proof (ExpressionLabelLowerBound p2); pose proof (ExpressionLabelLowerBound p3).
    remember latticeProof; destruct j; destruct OrdProof.
    pose proof (rel_trans _ _ _ l_rel_pc1 X) as low.
    pose proof (rel_trans _ _ _ l_rel_pc2 X0) as low0.

    assert (eq: l ≠ kpc ∧ l ≠ kpc0). {
      split.
      - destruct (level_eq_dec l kpc); auto. subst. pose proof (rel_antisym _ _ l_not_pc1 l_rel_pc1 X). contradiction.
      - destruct (level_eq_dec l kpc0); auto. subst. pose proof (rel_antisym _ _ l_not_pc2 l_rel_pc2 X0). contradiction.
    } destruct eq as [eq eq0].

    pose proof (ExpressionTimSec p1 p0).
    pose proof (ExpressionTimSec p2 p3).
    pose proof (DebranchPreservesMemEq p1_1 p2_1 memEq low eq low0 eq0) as memEq0.
    specialize (IHc1 _ _ _ _ _ _ _ _ _ _ _ _ _ p1_1 p2_1 memEq low eq low0 eq0).
    specialize (IHc2 _ _ _ _ _ _ _ _ _ _ _ _ _ p1_2 p2_2 memEq0 low eq low0 eq0).

    subst. reflexivity.

   + assert(k = kl). {
      Check @MemStoreEquivalenceImplExpressionEquivalence.
      pose proof (MemStoreEquivalenceImplExpressionEquivalence eProof p0 memEq).
      destruct H.
      - assumption.
      - contradiction.
    } subst. contradiction.

  + assert(k = kl). {

      pose proof (MemStoreEquivalenceImplExpressionEquivalence p1 eProof memEq).
```

```
destruct H.
- auto.
- contradiction.
} subst. contradiction.
+ assert (n' = n'0 ∧ k0 = k). {

pose proof (MemStoreEquivalenceImplExpressionEquivalence eProof eProof0 memEq).
destruct H.
- split; auto.
- contradiction.
} destruct H. subst.
pose proof (ExpressionTimSec eProof eProof0); subst.
destruct n'0;
try (specialize (IHc1 _ _ _ _ _ _ _ _ _ _ _ _ commandProof commandProof0
memEq l_rel_pc1 l_not_pc1 l_rel_pc2 l_not_pc2); subst; reflexivity );
try (specialize (IHc2 _ _ _ _ _ _ _ _ _ _ _ _ commandProof commandProof0
memEq l_rel_pc1 l_not_pc1 l_rel_pc2 l_not_pc2) ; subst; reflexivity).
-
assert (IHc': ∀ (n1 n2 : bool) (mu1 mu2 mu1' mu2' : MemStore) (l pc1 pc2 : Level)
(T1 T2 : TimingList),
@DebranchBigStep binop_eval rel latticeProof (Debranch c n1 l) mu1 pc1 T1
mu1' →
@DebranchBigStep binop_eval rel latticeProof(Debranch c n2 l) mu2 pc2 T2
mu2' →
@MemStoreObservationalEquivalent rel latticeProof mu1 l mu2 →
rel l pc1 → l ≠ pc1 → rel l pc2 → l ≠ pc2 → @MemStoreObservationalEquivalent
rel latticeProof mu1' l mu2').
{
clear. intros.
apply (DebranchPreservesMemEq X X0 X1 X2 H X3 H0).
}
pose proof (AlwaysLoopLengthDebranch p1); pose proof (AlwaysLoopLengthDebranch
p2); clear p1; clear p2.
destruct X as [x p1]. destruct X0 as [x0 p2]. pose proof (MemStoreEquivalen-
ceImplLoopLengthDebranchEq IHc' l_rel_pc1 l_not_pc1 l_rel_pc2 l_not_pc2 p1 p2 memEq);
subst x0.

revert memEq; generalize dependent mu2'; revert mu2; generalize dependent
mu1'; revert mu1; revert T2; revert T1.
dependent induction x; intros.
+ dependent destruction p1; dependent destruction p2. rewrite → (Expression-
TimSec expressionEvalProof expressionEvalProof0). reflexivity.
+ dependent destruction p1; dependent destruction p2.
```

```
        rewrite ← (ExpressionTimSec expressionEvalProof expressionEvalProof0).
        rewrite ← (IHc _ _ _ _ _ _ _ _ _ _ _ _ commandProof commandProof0 memEq
l_rel_pc1 l_not_pc1 l_rel_pc2 l_not_pc2).
        pose proof (IHc' _ _ _ _ _ _ _ _ _ _ _ _ commandProof commandProof0 memEq
l_rel_pc1 l_not_pc1 l_rel_pc2 l_not_pc2) as memEq'.
        rewrite ← (IHx _ _ _ _ p1 _ _ p2 memEq').
        reflexivity.
Qed.

Theorem CommandTimSec {binop_eval: BinOp → Primitive → Primitive → Primitive}
{rel: Level → Level → Type} {latticeProof: JoinSemilattice rel} :
  ∀ {c: Command} {mu1 mu2 mu1' mu2': MemStore} {pc: Level} {T1 T2: TimingList}
     (p1: @CommandBigStep binop_eval rel latticeProof c mu1 pc T1 mu1')
     (p2: @CommandBigStep binop_eval rel latticeProof c mu2 pc T2 mu2')
     (memEq: @MemStoreObservationalEquivalent rel latticeProof mu1 pc mu2),
    T1 = T2.
Proof.
  intros. dependent induction c.
  - dependent destruction p2; dependent destruction p1. reflexivity.
  - dependent destruction p2; dependent destruction p1. rewrite ← (ExpressionTimSec
eproof eproof0). reflexivity.
  - dependent destruction p2; dependent destruction p1.
    pose proof (CommandPreservesMemEq p1_1 p2_1 memEq) as memEq'.
    rewrite ← (IHc1 _ _ _ _ _ _ _ _ p1_1 p2_1 memEq).
    rewrite ← (IHc2 _ _ _ _ _ _ _ _ p1_2 p2_2 memEq').
    reflexivity.
  - dependent destruction p2; dependent destruction p1.
    + assert (notRel: pc ≠ kpc) by (apply not_eq_sym; apply (NotRelImplNotEq lattice-
Proof relProof)).
      assert (notRel0: pc ≠ kpc0) by (apply not_eq_sym; apply (NotRelImplNotEq latti-
ceProof relProof0)).
      assert (low: rel pc kpc) by (apply (ExpressionLabelLowerBound eProof)).
      assert (low0: rel pc kpc0) by (apply (ExpressionLabelLowerBound eProof0)).
      pose proof (DebranchPreservesMemEq debProof1 debProof0 memEq low notRel low0
notRel0) as memEq'.
      rewrite ← (DebranchTimSec debProof1 debProof0 memEq low notRel low0 notRel0).
      rewrite ← (DebranchTimSec debProof2 debProof3 memEq' low notRel low0 notRel0).
      rewrite ← (ExpressionTimSec eProof eProof0).
      reflexivity.
    + pose proof (ExpressionLabelLowestBound eProof relProof); subst k.
      assert (kpc = pc). {
        pose proof (MemStoreEquivalenceImplExpressionEquivalence eProof eProof0 memEq).
```

```
      dependent destruction H.
      - reflexivity.
      - pose proof (RelAlwaysRefl latticeProof l1High). contradiction.
    } subst. pose proof (RelAlwaysRefl latticeProof relProof0). contradiction.
  + pose proof (ExpressionLabelLowestBound eProof0 relProof0); subst k.
    assert (kpc = pc). {
      pose proof (MemStoreEquivalenceImplExpressionEquivalence eProof eProof0 memEq).
      dependent destruction H.
      - reflexivity.
      - pose proof (RelAlwaysRefl latticeProof l2High). contradiction.
    } subst. pose proof (RelAlwaysRefl latticeProof relProof). contradiction.

  + pose proof (ExpressionLabelLowestBound eProof relProof); subst k. pose proof (ExpressionLabelLowestBound eProof0 relProof0); subst k0.
    assert (n=n0). {
      pose proof (MemStoreEquivalenceImplExpressionEquivalence eProof eProof0 memEq).
      dependent destruction H.
      - reflexivity.
      - pose proof (RelAlwaysRefl latticeProof l1High). contradiction.
    } subst n0.
    rewrite ← (ExpressionTimSec eProof eProof0).
    destruct n;
    try (rewrite ← (IHc1 _ _ _ _ _ _ _ commandProof commandProof0 memEq); reflexivity);
      try (rewrite ← (IHc2 _ _ _ _ _ _ _ commandProof commandProof0 memEq); reflexivity).
- assert (IHc': ∀ (mu1 mu2 mu1' mu2' : MemStore) (pc : Level) (T1 T2 : TimingList),
      @CommandBigStep binop_eval rel latticeProof c mu1 pc T1 mu1' →
      @CommandBigStep binop_eval rel latticeProof c mu2 pc T2 mu2' →
      @MemStoreObservationalEquivalent rel latticeProof mu1 pc mu2 → @MemStoreObservationalEquivalent rel latticeProof mu1' pc mu2'). {
    clear; intros. apply (CommandPreservesMemEq X X0 X1).
  }

  pose proof (AlwaysLoopLengthCommand p1); pose proof (AlwaysLoopLengthCommand p2). clear p1 p2.
  destruct X as [x p1]. destruct X0 as [x0 p2]. pose proof (MemStoreEquivalenceImplLoopLengthCommandEq IHc' p1 p2 memEq). subst x0. generalize dependent memEq. generalize dependent mu2'; generalize dependent mu2; generalize dependent mu1'; generalize dependent mu1; revert T2; revert T1. dependent induction x; intros.

  + dependent destruction p2; dependent destruction p1.
    rewrite ← (ExpressionTimSec expressionEvalProof expressionEvalProof0).
    reflexivity.
```

```
  + dependent destruction p2; dependent destruction p1.
    pose proof (IHc' _ _ _ _ _ _ _ commandProof commandProof0 memEq) as memEq'.
    rewrite ← (ExpressionTimSec expressionProof expressionProof0).
    rewrite ← (IHc _ _ _ _ _ _ _ _ commandProof commandProof0 memEq).
    rewrite ← (IHx _ _ _ _ p1 _ _ p2 memEq').
    reflexivity.
Qed.
Compute True.
Definition NormalStore := Var → Primitive.
Definition StoreProjection (mu: MemStore) : NormalStore := fun x ⇒ fst (mu x).
Definition NormalUpdate (nu: NormalStore) (x: Var) (n: Primitive) : NormalStore :=
  fun x' ⇒ if (var_eq_dec x x') then n else (nu x').
Inductive ExpressionNormalBigStep {binop_eval: BinOp → Primitive → Primitive →
Primitive}: Expression → NormalStore → Primitive → Type :=
| NormalConstBigStep (prim: Primitive) (k: Level) (nu: NormalStore):
  ExpressionNormalBigStep (PrimitiveExpression prim k) nu prim

| NormalVarBigStep(x: Var) (nu: NormalStore)
  : ExpressionNormalBigStep (VarExpression x) nu (nu x)

| NormalOperBigStep (oper: BinOp) {nu: NormalStore}{e1 e2: Expression} {n1 n2: Primitive}
    (p1: ExpressionNormalBigStep e1 nu n1) (p2: ExpressionNormalBigStep e2 nu n2)
  : ExpressionNormalBigStep (BinOpExpression oper e1 e2) nu (binop_eval oper n1 n2).
Compute True.
Inductive NormalBigStep {binop_eval: BinOp → Primitive → Primitive → Primitive}
: Command → NormalStore → NormalStore → Type :=
| NormalSkipBigStep (nu: NormalStore)
  : NormalBigStep SkipCommand nu nu

| NormalSeqBigStep {c1 c2: Command} {nu nu' nu'': NormalStore}
    (p1: NormalBigStep c1 nu nu')
    (p2: NormalBigStep c2 nu' nu'')
  : NormalBigStep (SeqCommand c1 c2) nu nu''

| NormalWhileFBigStep {e: Expression} {nu: NormalStore} (c: Command) {n: Primitive}
    (expressionEvalProof: (@ExpressionNormalBigStep binop_eval e nu n ))
    (falseProof: n ≠ TruePrimitive)
  : NormalBigStep (WhileCommand e c) nu nu
```

```
| NormalWhileTBigStep {e: Expression} {nu nu' nu'': NormalStore} {c: Command}
      (expressionEvalProof: (@ExpressionNormalBigStep binop_eval e nu TruePrimitive))
      (commandProof: NormalBigStep c nu nu')
      (restLoopProof: NormalBigStep (WhileCommand e c) nu' nu'')
  : NormalBigStep (WhileCommand e c) nu nu''

| NormalAssnBigStep {e: Expression} {nu: NormalStore} {x: Var} {n: Primitive}
     (eproof: @ExpressionNormalBigStep binop_eval e nu n)
  : NormalBigStep (AssnCommand x e) nu (NormalUpdate nu x n)

| NormalIfBigStep
     {e: Expression} {nu nu': NormalStore} {n: Primitive} (c1 c2: Command)
     (eProof: @ExpressionNormalBigStep binop_eval e nu n )
     (commandProof: let c := match n with | TruePrimitive ⇒ c1 | _ ⇒ c2 end in
                    NormalBigStep c nu nu')
  : NormalBigStep (IfCommand e c1 c2) nu nu'.
Compute True.

Inductive LoopLengthNormal {binop_eval: BinOp → Primitive → Primitive → Primitive} : NormalStore → Expression → Command → NormalStore → nat → Type :=
| LoopLengthNormal0 {mu: NormalStore} {e: Expression} {n: Primitive}
     (c: Command)
     (expressionEvalProof: @ExpressionNormalBigStep binop_eval e mu n)
     (primProof: n ≠ TruePrimitive)
  : LoopLengthNormal mu e c mu 0

| LoopLengthNormalSn {mu mu' mu'': NormalStore} {e: Expression} {n: nat} {c: Command}
     (expressionProof: @ExpressionNormalBigStep binop_eval e mu TruePrimitive)
     (commandProof: @NormalBigStep binop_eval c mu mu')
     (whileProof: @NormalBigStep binop_eval (WhileCommand e c) mu' mu'')
     (indProof: LoopLengthNormal mu' e c mu'' n)
  : LoopLengthNormal mu e c mu'' (S n).

Lemma AlwaysLoopLengthNormal {binop_eval: BinOp → Primitive → Primitive → Primitive}:
  ∀ {e: Expression} {c: Command} {mu mu': NormalStore},
     @NormalBigStep binop_eval (WhileCommand e c) mu mu' →
     EX (fun n ⇒ @LoopLengthNormal binop_eval mu e c mu' n).
Proof.
  intros.
  dependent induction H.
  - apply (EX_intro _ 0 (LoopLengthNormal0 c expressionEvalProof falseProof)).
  - clear IHNormalBigStep1. assert (WhileCommand e c = WhileCommand e c) by auto.
```

```
specialize (IHNormalBigStep2 _ _ H1); destruct IHNormalBigStep2. apply (EX_intro _
(S x) (LoopLengthNormalSn expressionEvalProof H H0 l)).
Qed.

Theorem ExpressionSystemEquivalence {binop_eval: BinOp → Primitive → Primitive →
Primitive} {rel: Level → Level → Type} {latticeProof: JoinSemilattice rel}:
  ∀ {e: Expression} {mu: MemStore} {T: TimingList} {pc: Level} {n1 n2: Primitive} {k:
Level},
    @ExpressionBigStep binop_eval rel latticeProof e mu pc T n1 k →
    @ExpressionNormalBigStep binop_eval e (StoreProjection mu) n2 →
    n1 = n2.
Proof.
  intros e mu T pc n1 n2 k. intros eProof. intros nProof. dependent induction e;
dependent destruction eProof; dependent destruction nProof.
  - reflexivity.
  - unfold StoreProjection. reflexivity.
  - specialize (IHe1 _ _ _ _ _ _ eProof1 nProof1); specialize (IHe2 _ _ _ _ _ _ eProof2
nProof2). subst. reflexivity.
Qed.

Lemma DebranchNormalLoopEq {binop_eval: BinOp → Primitive → Primitive → Primitive}
{rel: Level → Level → Type} {latticeProof: JoinSemilattice rel}:
  ∀ {e: Expression} {c: Command} {mu mu': MemStore} {nu: NormalStore} {pc l:
Level} {T: TimingList} {n1 n2: nat}
    (cEq: ∀ (nu : NormalStore) (mu mu' : MemStore) (l pc : Level) (T : TimingList),
        @DebranchBigStep binop_eval rel latticeProof (Debranch c true l) mu pc T mu'
→
        @NormalBigStep binop_eval c (StoreProjection mu) nu → StoreProjection mu' =
nu),
    @LoopLengthDebranch binop_eval rel latticeProof pc mu e c true l T mu' n1 →
    @LoopLengthNormal binop_eval (StoreProjection mu) e c nu n2 →
    n1 = n2.
Proof.
  intros. generalize dependent mu. generalize dependent mu'. generalize dependent
nu. revert T. dependent induction n1; intros; dependent destruction X; dependent
destruction H.
    + reflexivity.
    + pose proof (ExpressionSystemEquivalence expressionEvalProof expressionProof). destruct
H; contradiction.
    + pose proof (ExpressionSystemEquivalence expressionEvalProof expressionEvalProof0).
destruct H; subst; contradiction.
    + pose proof (cEq _ _ _ _ _ _ commandProof commandProof0).
      apply f_equal.
      apply (IHn1 _ cEq Tw mu''0 mu'' mu'). apply X. rewrite H0. apply H.
```

```
Qed.

From Stdlib Require Import Logic.FunctionalExtensionality.
From Stdlib Require Import Bool.Bool.

Lemma DebranchFalseIdent {binop_eval: BinOp → Primitive → Primitive → Primitive}
{rel: Level → Level → Type} {latticeProof: JoinSemilattice rel}:
  ∀ {c: Command} {mu mu': MemStore} {l pc: Level} {T: TimingList}
     (p: @DebranchBigStep binop_eval rel latticeProof (Debranch c false l) mu pc T
mu'),
    StoreProjection mu = StoreProjection mu'.
Proof.
  intros c. intros. dependent induction c.
  - dependent destruction p. simpl. reflexivity.
  - dependent destruction p. unfold StoreProjection. apply functional_extensionality.
intros x'. unfold MemUpdate; unfold t_update. destruct (var_eq_dec x x').
    + simpl. subst. reflexivity.
    + reflexivity.
  - dependent destruction p.
    specialize (IHc1 _ _ _ _ _ p1).
    specialize (IHc2 _ _ _ _ _ p2).
    rewrite → IHc1.
    rewrite → IHc2.
    reflexivity.
  - dependent destruction p.
    + rewrite andb_false_r in p3. rewrite andb_false_r in p4.
      specialize (IHc1 _ _ _ _ _ p3).
      specialize (IHc2 _ _ _ _ _ p4).
      rewrite IHc1. rewrite IHc2.
      reflexivity.
    + destruct n'.
      ++ specialize (IHc1 _ _ _ _ _ commandProof). rewrite IHc1. reflexivity.
      ++ specialize (IHc2 _ _ _ _ _ commandProof). rewrite IHc2. reflexivity.
      ++ specialize (IHc2 _ _ _ _ _ commandProof). rewrite IHc2. reflexivity.
  - pose proof (AlwaysLoopLengthDebranch p). destruct X as [num LOOP]. generalize
dependent mu. generalize dependent mu'. revert T. induction num; intros.
    + dependent destruction LOOP. reflexivity.
    + dependent destruction LOOP. specialize (IHc _ _ _ _ _ commandProof). specialize
(IHnum _ _ _ restLoopProof LOOP). rewrite → IHc. rewrite IHnum. reflexivity.
Qed.

Lemma DebranchSystemEquivalence {binop_eval: BinOp → Primitive → Primitive → Prim-
itive} {rel: Level → Level → Type} {latticeProof: JoinSemilattice rel}:
  ∀ {c: Command} {nu: NormalStore} {mu mu': MemStore} {l pc: Level} {T: TimingList}
     (p: @DebranchBigStep binop_eval rel latticeProof (Debranch c true l) mu pc T mu')
```

```
    (np: @NormalBigStep binop_eval c (StoreProjection mu) nu),
   (StoreProjection mu') = nu.

Proof.
intros c. dependent induction c; intros.
  - dependent destruction np; dependent destruction p. reflexivity.
  - dependent destruction np; dependent destruction p. pose proof (ExpressionSystemEquivalence evalProof eproof) as EQ. rewrite <- EQ. unfold NormalUpdate; unfold MemUpdate; unfold t_update. unfold StoreProjection. apply functional_extensionality. intros x0. destruct (var_eq_dec x x0).
     + simpl. reflexivity.
     + unfold StoreProjection. reflexivity.
  - dependent destruction np; dependent destruction p. specialize (IHc1 _ _ _ _ _ _ p1 np1). subst; specialize (IHc2 _ _ _ _ _ _ p2 np2). apply IHc2.
  - dependent destruction p; dependent destruction np.
     + pose proof (ExpressionSystemEquivalence p1 eProof); pose proof (ExpressionSystemEquivalence p2 eProof). subst. clear H0.
       destruct n.
       ++ simpl in p4. simpl in p3. pose proof (DebranchFalseIdent p4). rewrite <- H. specialize (IHc1 _ _ _ _ _ _ p3 commandProof). rewrite IHc1. reflexivity.
       ++ simpl in p4. simpl in p3. pose proof (DebranchFalseIdent p3). apply (IHc2 _ mu' _ l kpc T4). apply p4. rewrite <- H. apply commandProof.
       ++ simpl in p4. simpl in p3. pose proof (DebranchFalseIdent p3). apply (IHc2 _ mu' _ l kpc T4). apply p4. rewrite <- H. apply commandProof.
     + pose proof (ExpressionSystemEquivalence eProof eProof0); subst.
       destruct n.
       ++ specialize (IHc1 _ _ _ _ _ _ commandProof commandProof0); assumption.
       ++ specialize (IHc2 _ _ _ _ _ _ commandProof commandProof0); assumption.
       ++ specialize (IHc2 _ _ _ _ _ _ commandProof commandProof0); assumption.
  - pose proof (AlwaysLoopLengthDebranch p). pose proof (AlwaysLoopLengthNormal np). destruct X; destruct H. pose proof (DebranchNormalLoopEq IHc l0 l1). subst x0. generalize dependent mu. revert nu. revert mu'. revert T. induction x; intros.
     + dependent destruction l0; dependent destruction l1. reflexivity.
     + dependent destruction l0; dependent destruction l1. pose proof (IHc _ _ _ _ _ _ commandProof commandProof0). apply (IHx Tw mu'' mu''0 mu'). assumption. rewrite H. assumption. assumption. rewrite H. assumption.
Qed.

Lemma CommandNormalLoopEq {binop_eval: BinOp -> Primitive -> Primitive -> Primitive} {rel: Level -> Level -> Type} {latticeProof: JoinSemilattice rel}:
  ∀ {e: Expression} {c: Command} {mu mu': MemStore} {nu: NormalStore} {pc: Level} {T: TimingList} {n1 n2: nat}
    (cEq: ∀ (mu mu' : MemStore) (nu : NormalStore) (T : TimingList) (pc : Level),
```

@**CommandBigStep** *binop_eval rel latticeProof c mu pc T mu'* →
@**NormalBigStep** *binop_eval c* (StoreProjection *mu*) *nu* → StoreProjection *mu'* = *nu*),
@**LoopLengthCommand** *binop_eval rel latticeProof pc mu e c T mu' n1* →
@**LoopLengthNormal** *binop_eval* (StoreProjection *mu*) *e c nu n2* →
n1 = *n2*.

Proof.

intros. generalize dependent *mu*. generalize dependent *mu'*. generalize dependent *nu*. *revert T*. dependent induction *n1*; intros; dependent destruction *X*; dependent destruction *H*.

+ reflexivity.

+ pose *proof* (ExpressionSystemEquivalence *expressionEvalProof expressionProof*). destruct *H*; *contradiction*.

+ pose *proof* (ExpressionSystemEquivalence *expressionProof expressionEvalProof*). destruct *H*; subst; *contradiction*.

+ pose *proof* (*cEq _ _ _ _ _ commandProof commandProof0*).
apply f_equal.
apply (*IHn1 _ cEq Tw mu''0 mu'' mu'*). apply *X*. rewrite *H0*. apply *H*.

Qed.

Theorem CommandSystemEquivalence {*binop_eval*: **BinOp** → **Primitive** → **Primitive** → **Primitive**} {*rel*: **Level** → **Level** → Type} {*latticeProof*: **JoinSemilattice** *rel*}:
∀ {*c*: **Command**} {*mu mu'*: MemStore} {*nu*: NormalStore} {*T*: TimingList} {*pc*: **Level**},
@**CommandBigStep** *binop_eval rel latticeProof c mu pc T mu'* →
@**NormalBigStep** *binop_eval c* (StoreProjection *mu*) *nu* →
(StoreProjection *mu'*) = *nu*.

Proof.

intros *c mu mu' nu T pc*. intros *cProof*. intros *nProof*. dependent induction *c*.

- dependent destruction *cProof*; dependent destruction *nProof*. unfold StoreProjection. reflexivity.

- dependent destruction *cProof*; dependent destruction *nProof*. pose *proof* (ExpressionSystemEquivalence *eproof eproof0*) as *EQ*. rewrite ← *EQ*. unfold NormalUpdate; unfold MemUpdate; unfold t_update. unfold StoreProjection. apply functional_extensionality. intros *x0*. destruct (var_eq_dec *x x0*).

+ simpl. reflexivity.

+ unfold StoreProjection. reflexivity.

- dependent destruction *cProof*; dependent destruction *nProof*. specialize (*IHc1 _ _ _ _ _ cProof1 nProof1*). subst; specialize (*IHc2 _ _ _ _ _ cProof2 nProof2*). apply *IHc2*.

- dependent destruction *cProof*; dependent destruction *nProof*.

+ pose *proof* (ExpressionSystemEquivalence *eProof eProof0*); subst.
destruct *n1*; simpl in *debProof1* ; simpl in *debProof2*.
++ pose *proof* (DebranchFalseIdent *debProof2*). rewrite ← *H*. apply (DebranchSys-

temEquivalence *debProof1 commandProof*).

++ pose *proof* (DebranchFalseIdent *debProof1*). rewrite *H* in *commandProof*. apply (DebranchSystemEquivalence *debProof2 commandProof*).

++ pose *proof* (DebranchFalseIdent *debProof1*). rewrite *H* in *commandProof*. apply (DebranchSystemEquivalence *debProof2 commandProof*).

+ pose *proof* (ExpressionSystemEquivalence *eProof eProof0*); subst *n0*.
destruct *n*.

++ specialize (*IHc1* _ _ _ _ _ *commandProof commandProof0*); assumption.

++ specialize (*IHc2* _ _ _ _ _ *commandProof commandProof0*); assumption.

++ specialize (*IHc2* _ _ _ _ _ *commandProof commandProof0*); assumption.

- pose *proof* (AlwaysLoopLengthCommand *cProof*). pose *proof* (AlwaysLoopLengthNormal *nProof*). destruct *X*; destruct *H*. pose *proof* (CommandNormalLoopEq *IHc l l0*). subst *x0*. generalize dependent *mu*. *revert nu*. *revert mu'*. *revert T*. induction *x*; intros.

+ dependent destruction *l*; dependent destruction *l0*. reflexivity.

+ dependent destruction *l*; dependent destruction *l0*. pose *proof* (*IHc* _ _ _ _ _ *commandProof commandProof0*). apply (*IHx Tw mu'' mu''0 mu'*). assumption. rewrite *H*. assumption. assumption. rewrite *H*. assumption.

Qed.