GraalVM Stamp Theory

April 17, 2024

Abstract

The GraalVM compiler uses stamps to track type and range information during program analysis. Type information is recorded by using distinct subclasses of the abstract base class `Stamp`, i.e. `IntegerStamp` is used to represent an integer type. Each subclass introduces facilities for tracking range information. Every subclass of the `Stamp` class forms a lattice, together with an arbitrary top and bottom element each sublattice forms a lattice of all stamps. This Isabelle/HOL theory models stamps as instantiations of a lattice.

Contents

1 Stamps: Type and Range Information

theory *StampLattice*
imports
Values
HOL.Lattices
begin

1.1 Void Stamp

The VoidStamp represents a type with no associated values. The VoidStamp lattice is therefore a simple single element lattice.

datatype *void* =
VoidStamp

instantiation *void* :: *order*
begin

definition *less-eq-void* :: *void* ⇒ *void* ⇒ *bool* **where**
less-eq-void a b = *True*

definition *less-void* :: *void* ⇒ *void* ⇒ *bool* **where**
less-void a b = *False*

instance
apply *standard*
apply (*simp add*: *less-eq-void-def less-void-def*)+
by (*metis* (*full-types*) *void.exhaust*)

end

instantiation *void* :: *semilattice-inf*
begin

definition *inf-void* :: *void* ⇒ *void* ⇒ *void* **where**
inf-void a b = *VoidStamp*

instance
apply *standard*
by (*simp add*: *less-eq-void-def*)+

end

instantiation *void* :: *semilattice-sup*
begin

definition *sup-void* :: *void* ⇒ *void* ⇒ *void* **where**
sup-void a b = *VoidStamp*

instance
  apply *standard*
  by (*simp add*: *less-eq-void-def*)+

end

instantiation *void* :: *bounded-lattice*
begin

definition *bot-void* :: *void* **where**
  bot-void = *VoidStamp*

definition *top-void* :: *void* **where**
  top-void = *VoidStamp*

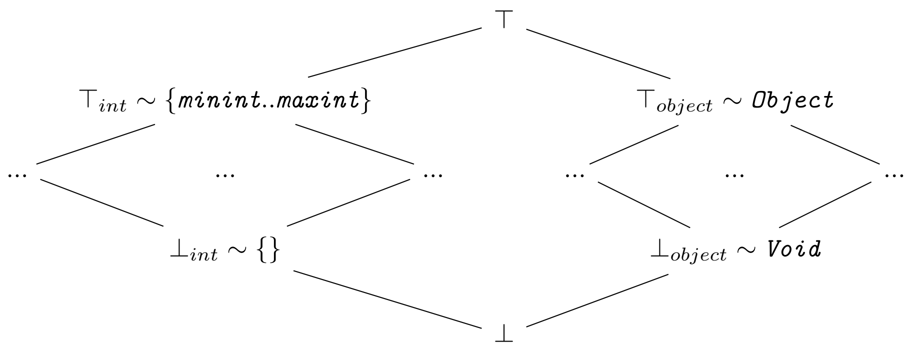
instance
  apply *standard*
  by (*simp add*: *less-eq-void-def*)+

end

Definition of the stamp type

datatype *stamp* =
  intstamp int64 int64 — Type: Integer; Range: Lower Bound & Upper Bound

1.2 Stamp Lattice



1.2.1 Stamp Order

Defines an ordering on the stamp type.

One stamp is less than another if the valid values for the stamp are a strict subset of the other stamp.

instantiation *stamp* :: *order*
begin

fun *less-eq-stamp* :: *stamp* ⇒ *stamp* ⇒ *bool* **where**
less-eq-stamp (*intstamp l1 u1*) (*intstamp l2 u2*) = ({*l1*..*u1*} ⊆ {*l2*..*u2*})

fun *less-stamp* :: *stamp* ⇒ *stamp* ⇒ *bool* **where**
less-stamp (*intstamp l1 u1*) (*intstamp l2 u2*) = ({*l1*..*u1*} ⊂ {*l2*..*u2*})

lemma *less-le-not-le*:
fixes *x y* :: *stamp*
shows ($x < y$) = ($x \leq y \wedge \neg\, y \leq x$)
by (*metis subset-not-subset-eq stamp.exhaust less-stamp.simps less-eq-stamp.simps*)

lemma *order-refl*:
fixes *x* :: *stamp*
shows $x \leq x$
by (*metis stamp.exhaust dual-order.refl less-eq-stamp.simps*)

lemma *order-trans*:
fixes *x y z* :: *stamp*
shows $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$
proof −
fix *x* :: *stamp* **and** *y* :: *stamp* **and** *z* :: *stamp*
assume $x \leq y$
assume $y \leq z$
obtain *l1 u1* **where** *xdef*: *x* = *intstamp l1 u1*
using *stamp.exhaust* **by** *auto*
obtain *l2 u2* **where** *ydef*: *y* = *intstamp l2 u2*
using *stamp.exhaust* **by** *auto*
obtain *l3 u3* **where** *zdef*: *z* = *intstamp l3 u3*
using *stamp.exhaust* **by** *auto*
have *s1*: {*l1*..*u1*} ≤ {*l2*..*u2*}
using ‹$x \leq y$› **by** (*simp add*: *ydef xdef*)
have *s2*: {*l2*..*u2*} ≤ {*l3*..*u3*}
using ‹$y \leq z$› **by** (*simp add*: *zdef ydef*)
from *s1 s2* **have** {*l1*..*u1*} ≤ {*l3*..*u3*}
by (*meson dual-order.trans*)
then show $x \leq z$
by (*simp add*: *zdef xdef*)
qed

lemma *antisym*:
fixes *x y* :: *stamp*
shows $x \leq y \Longrightarrow y \leq x \Longrightarrow x = y$
proof −
fix *x* :: *stamp*
fix *y* :: *stamp*
assume *xlessy*: $x \leq y$
assume *ylessx*: $y \leq x$
obtain *l1 u1* **where** *xdef*: *x* = *intstamp l1 u1*
using *stamp.exhaust* **by** *auto*

```
  obtain l2 u2 where ydef: y = intstamp l2 u2
    using stamp.exhaust by auto
  from xlessy have s1: {l1..u1} ⊆ {l2..u2}
    by (simp add: ydef xdef)
  from ylessx have s2: {l2..u2} ⊆ {l1..u1}
    by (simp add: ydef xdef)
  have {l1..u1} ⊆ {l2..u2} ⟹ {l2..u2} ⊆ {l1..u1} ⟹ {l1..u1} = {l2..u2}
    by auto
  then have s3: {l1..u1} = {l2..u2} ⟹ (l1 = l2) ∧ (u1 = u2)


    sorry
  then have (l1 = l2) ∧ (u1 = u2) ⟹ x = y
    using xdef ydef by fastforce
  then show x = y
    using s1 s2 s3 by fastforce
qed

instance
  apply standard
  by (simp add: antisym order-trans order-refl less-le-not-le)+
end
```

1.2.2 Stamp Join

Defines the *join* operation for stamps.

For any two stamps, the *join* is defined as the intersection of the valid values for the stamp.

```
instantiation stamp :: semilattice-inf
begin

notation inf (infix ⊓ 65)

fun inf-stamp :: stamp ⇒ stamp ⇒ stamp where
  inf-stamp (intstamp l1 u1) (intstamp l2 u2) = intstamp (max l1 l2) (min u1 u2)

lemma inf-le1:
  fixes x y :: stamp
  shows (x ⊓ y) ≤ x
proof –
  fix x :: stamp
  fix y :: stamp
  obtain l1 u1 where xdef: x = intstamp l1 u1
    using stamp.exhaust by auto
  obtain l2 u2 where ydef: y = intstamp l2 u2
    using stamp.exhaust by auto
  have joindef: x ⊓ y = intstamp (max l1 l2) (min u1 u2)
    (is ?join = intstamp ?l3 ?u3)
```

```
    by (simp add: ydef xdef)
  have leq: {?l3..?u3} ⊆ {l1..u1}
    by simp
  have (x ⊓ y) ≤ x = ({?l3..?u3} ⊆ {l1..u1})
    using joindef by (simp add: xdef)
  then show (x ⊓ y) ≤ x
    by (simp add: leq)
qed

lemma inf-le2:
  fixes x y :: stamp
  shows (x ⊓ y) ≤ y
proof –
  fix x :: stamp
  fix y :: stamp
  obtain l1 u1 where xdef: x = intstamp l1 u1
    using stamp.exhaust by auto
  obtain l2 u2 where ydef: y = intstamp l2 u2
    using stamp.exhaust by auto
  have joindef: x ⊓ y = intstamp (max l1 l2) (min u1 u2)
    (is ?join = intstamp ?l3 ?u3)
    by (simp add: ydef xdef)
  have leq: {?l3..?u3} ⊆ {l2..u2}
    by simp
  have (x ⊓ y) ≤ y = ({?l3..?u3} ⊆ {l2..u2})
    using joindef by (simp add: ydef)
  then show (x ⊓ y) ≤ y
    by (simp add: leq)
qed

lemma inf-greatest:
  fixes x y z :: stamp
  shows x ≤ y ⟹ x ≤ z ⟹ x ≤ (y ⊓ z)
proof –
  fix x y z :: stamp
  assume xlessy: x ≤ y
  assume xlessz: x ≤ z
  obtain l1 u1 where xdef: x = intstamp l1 u1
    using stamp.exhaust by auto
  obtain l2 u2 where ydef: y = intstamp l2 u2
    using stamp.exhaust by auto
  obtain l3 u3 where zdef: z = intstamp l3 u3
    using stamp.exhaust by auto
  obtain l4 u4 where yzdef: y ⊓ z = intstamp l4 u4
    by (meson inf-stamp.elims)
  have max4: l4 = max l2 l3
    using yzdef by (simp add: zdef ydef)
  have min4: u4 = min u2 u3
    using yzdef by (simp add: zdef ydef)
```

```
  have {l1..u1} ⊆ {l2..u2}
    using xlessy by (simp add: ydef xdef)
  have {l1..u1} ⊆ {l3..u3}
    using xlessz by (simp add: zdef xdef)
  have leq: {l1..u1} ⊆ {l4..u4}
    using ‹{l1..u1} ⊆ {l2..u2}› ‹{l1..u1} ⊆ {l3..u3}› by (simp add: min4 max4)
  have x ≤ (y ⊓ z) = ({l1..u1} ⊆ {l4..u4})
    by (simp add: xdef yzdef)
  then show x ≤ (y ⊓ z)
    using leq by simp
qed

instance
  apply standard
  by (simp add: inf-greatest inf-le2 inf-le1)+
end
```

1.2.3 Stamp Meet

Defines the *meet* operation for stamps.

For any two stamps, the *meet* is defined as the union of the valid values for the stamp.

```
instantiation stamp :: semilattice-sup
begin

notation sup (infix ⊔ 65)

fun sup-stamp :: stamp ⇒ stamp ⇒ stamp where
  sup-stamp (intstamp l1 u1) (intstamp l2 u2) = intstamp (min l1 l2) (max u1 u2)

lemma sup-ge1:
  fixes x y :: stamp
  shows x ≤ x ⊔ y
proof −
  fix x :: stamp
  fix y :: stamp
  obtain l1 u1 where xdef: x = intstamp l1 u1
    using stamp.exhaust by auto
  obtain l2 u2 where ydef: y = intstamp l2 u2
    using stamp.exhaust by auto
  have joindef: x ⊔ y = intstamp (min l1 l2) (max u1 u2)
    (is ?join = intstamp ?l3 ?u3)
    by (simp add: ydef xdef)
  have leq: {l1..u1} ⊆ {?l3..?u3}
    by simp
  have x ≤ x ⊔ y = ({l1..u1} ⊆ {?l3..?u3})
    using joindef by (simp add: xdef)
  then show x ≤ x ⊔ y
```

```
    by (simp add: leq)
qed

lemma sup-ge2:
  fixes x y :: stamp
  shows y ≤ x ⊔ y
proof −
  fix x :: stamp
  fix y :: stamp
  obtain l1 u1 where xdef: x = intstamp l1 u1
    using stamp.exhaust by auto
  obtain l2 u2 where ydef: y = intstamp l2 u2
    using stamp.exhaust by auto
  have joindef: x ⊔ y = intstamp (min l1 l2) (max u1 u2)
    (is ?join = intstamp ?l3 ?u3)
    by (simp add: ydef xdef)
  have leq: {l2..u2} ⊆ {?l3..?u3} (is ?subset-thesis)
    by simp
  have ?thesis = (?subset-thesis)
     by (metis StampLattice.sup-ge1 max.commute min.commute sup-stamp.elims
less-eq-stamp.simps
        sup-stamp.simps)
  then show ?thesis
    by simp
qed

lemma sup-least:
  fixes x y z :: stamp
  shows y ≤ x ⟹ z ≤ x ⟹ ((y ⊔ z) ≤ x)
proof −
  fix x y z :: stamp
  assume xlessy: y ≤ x
  assume xlessz: z ≤ x
  obtain l1 u1 where xdef: x = intstamp l1 u1
    using stamp.exhaust by auto
  obtain l2 u2 where ydef: y = intstamp l2 u2
    using stamp.exhaust by auto
  obtain l3 u3 where zdef: z = intstamp l3 u3
    using stamp.exhaust by auto
  have yzdef: y ⊔ z = intstamp (min l2 l3) (max u2 u3)
    (is ?meet = intstamp ?l4 ?u4)
    by (simp add: ydef zdef)
  have s1: {l2..u2} ⊆ {l1..u1}
    using xlessy by (simp add: ydef xdef)
  have s2: {l3..u3} ⊆ {l1..u1}
    using xlessz by (simp add: zdef xdef)
  have leq: {?l4..?u4} ⊆ {l1..u1} (is ?subset-thesis)

    by (metis (no-types, opaque-lifting) inf.orderE inf-stamp.simps max.bounded-iff
```

max.cobounded2
min.bounded-iff min.cobounded2 stamp.inject xdef xlessy ydef zdef atLeastat-Most-subset-iff
xlessz)
have ($y \sqcup z \leq x$) = *?subset-thesis*
by (*simp add*: *xdef yzdef*)
then show ($y \sqcup z \leq x$)
using *leq* **by** *simp*
qed

instance
apply *standard*
by (*simp add*: *sup-least sup-ge2 sup-ge1*)+
end

1.2.4 Stamp Bounds

Defines the top and bottom elements of the stamp lattice.

This poses an interesting question as our stamp type is a union of the various *Stamp* subclasses, e.g. *IntegerStamp*, *ObjectStamp*, etc.

Each subclass should preferably have its own unique top and bottom element, i.e. An *IntegerStamp* would have the top element of the full range of integers allowed by the bit width and a bottom of a range with no integers. While the *ObjectStamp* should have *Object* as the top and *Void* as the bottom element.

instantiation *stamp* :: *bounded-lattice*
begin

notation *bot* (⊥ *50*)
notation *top* (⊤ *50*)

definition *width-min* :: *nat* ⇒ *int64* **where**
width-min bits = −(*2*^(*bits*−*1*))

definition *width-max* :: *nat* ⇒ *int64* **where**
width-max bits = (*2*^(*bits*−*1*)) − *1*

value (*sint* (*width-min 64*), *sint* (*width-max 64*))
value *max-word*::*int64*

lemma
assumes *x* = *width-min 64*
assumes *y* = *width-max 64*
shows *sint x* < *sint y*
by (*simp add*: *assms width-max-def width-min-def*)

Note that this definition is valid for unsigned integers only.

The bottom and top element for signed integers would be (- 9223372036854775808, 9223372036854775807).

For unsigned we have (0, 18446744073709551615).

For Java we are likely to be more concerned with signed integers. To use the appropriate bottom and top for signed integers we would need to change our definition of less_eq from l1..u1 <= l2..u2 to sint l1..sint u1 <= sint l2..sint u2

We may still find an unsigned integer stamp useful. I plan to investigate the Java code to see if this is useful and then apply the changes to switch to signed integers.

definition *bot-stamp* = *intstamp* (−*1*) *0*
definition *top-stamp* = *intstamp* *0* (−*1*)

lemma *bot-least*:
  fixes *a* :: *stamp*
  shows (⊥) ≤ *a*
proof −
  obtain *min max* **where** *bot-def*:⊥ = *intstamp max min*
    by (*simp add*: *bot-stamp-def*)
  have *min* < *max*
    using *bot-def word-gt-0* **unfolding** *bot-stamp-def* **by** *fastforce*
  then have {*max..min*} = {}
    by (*simp add*: *bot-def*)
  then show *?thesis*
    using *less-eq-stamp.simps* **by** (*simp add*: *stamp.induct bot-stamp-def*)
qed

lemma *top-greatest*:
  fixes *a* :: *stamp*
  shows *a* ≤ (⊤)
proof −
  obtain *min max* **where** *top-def*:⊤ = *intstamp min max*
    by (*simp add*: *top-stamp-def*)
  have *max-is-max*: ¬(∃ *n*. *n* > *max*)
    by (*metis stamp.inject top-def top-stamp-def word-order.extremum-strict*)
  have *min-is-min*: ¬(∃ *n*. *n* < *min*)
   by (*metis not-less-iff-gr-or-eq stamp.inject top-def top-stamp-def word-coorder.not-eq-extremum*)
  have ¬(∃ *l u*. {*min..max*} < {*l..u*})
    by (*metis atLeastatMost-psubset-iff not-less min-is-min max-is-max*)
  then show *?thesis*
    unfolding *top-stamp-def* **using** *less-eq-stamp.elims*(*3*) **by** *fastforce*
qed

instance
  apply *standard*
  by (*simp add*: *top-greatest bot-least*)+
end

1.3 Java Stamp Methods

The following are methods from the Java Stamp class, they are the methods primarily used for optimizations.

definition *is-unrestricted* :: *stamp* ⇒ *bool* **where**
is-unrestricted s = (⊤ = *s*)

fun *is-empty* :: *stamp* ⇒ *bool* **where**
is-empty s = (⊥ = *s*)

fun *as-constant* :: *stamp* ⇒ *Value option* **where**
as-constant (*intstamp l u*) = (*if* (*card* {*l..u*}) = *1*
then Some (*IntVal 64* (*SOME x. x* ∈ {*l..u*}))
else None)

definition *always-distinct* :: *stamp* ⇒ *stamp* ⇒ *bool* **where**
always-distinct stamp1 stamp2 = (⊥ = (*stamp1* ⊓ *stamp2*))

definition *never-distinct* :: *stamp* ⇒ *stamp* ⇒ *bool* **where**
never-distinct stamp1 stamp2 =
(*as-constant stamp1* = *as-constant stamp2* ∧ *as-constant stamp1* ≠ *None*)

1.4 Mapping to Values

fun *valid-value* :: *stamp* => *Value* => *bool* **where**
valid-value (*intstamp l u*) (*IntVal b v*) = (*v* ∈ {*l..u*}) |
valid-value (*intstamp l u*) - = *False*

The *valid-value* function is used to map a stamp instance to the values that are allowed by the stamp.

It would be nice if there was a slightly more integrated way to perform this mapping as it requires some infrastructure to prove some fairly simple properties.

lemma *bottom-range-empty*:
¬(*valid-value* (⊥) *v*)
unfolding *bot-stamp-def* **using** *valid-value.elims*(*2*) **by** *fastforce*

lemma *join-values*:
assumes *joined* = *x-stamp* ⊓ *y-stamp*
shows *valid-value joined x* ⟷ (*valid-value x-stamp x* ∧ *valid-value y-stamp x*)
proof (*cases x*)
case *UndefVal*
then show *?thesis*
using *valid-value.elims*(*2*) **by** *auto*

next
case (*IntVal b x3*)
obtain *lx ux* **where** *xdef*: *x-stamp* = *intstamp lx ux*

```
    using stamp.exhaust by auto
  obtain ly uy where ydef: y-stamp = intstamp ly uy
    using stamp.exhaust by auto
  obtain v where x = IntVal b v
    by (simp add: IntVal)
  have joined = intstamp (max lx ly) (min ux uy)
    (is joined = intstamp ?lj ?uj)
    by (simp add: xdef ydef assms)
  then have valid-value joined (IntVal b v) = (v ∈ {?lj..?uj})
    by simp
  then show ?thesis
    using ‹x = IntVal b v› by (auto simp add: ydef xdef)
next
  case (ObjRef x5)
  then show ?thesis
    using valid-value.elims(2) by auto
next
  case (ObjStr x6)
  then show ?thesis
    using valid-value.elims(2) by auto
next
  case (ArrayVal x51 x52)
  then show ?thesis
    using valid-value.elims(2) by blast
qed

lemma disjoint-empty:
  fixes x-stamp y-stamp :: stamp
  assumes ⊥ = x-stamp ⊓ y-stamp
  shows ¬(valid-value x-stamp x ∧ valid-value y-stamp x)
  using bottom-range-empty by (simp add: join-values assms)

experiment begin
```

A possible equivalent alternative to the definition of less_eq

```
fun less-eq-alt :: 'a::ord × 'a ⇒ 'a × 'a ⇒ bool where
  less-eq-alt (l1, u1) (l2, u2) = ((¬ l1 ≤ u1) ∨ l2 ≤ l1 ∧ u1 ≤ u2)
```

Proof equivalence

```
lemma
  fixes l1 l2 u1 u2 :: int
  assumes l1 ≤ u1 ∧ l2 ≤ u2
  shows {l1..u1} ⊆ {l2..u2} = ((l1 ≥ l2) ∧ (u1 ≤ u2))
  by (simp add: assms)

lemma
  fixes l1 l2 u1 u2 :: int
  shows {l1..u1} ⊆ {l2..u2} = less-eq-alt (l1, u1) (l2, u2)
  by simp
```

end

1.5 Generic Integer Stamp

Experimental definition of integer stamps generically, restricting the datatype to only allow valid ranges and the bottom integer element (max_int..min_int).

lemma
  assumes $(x{::}int) > 0$
  shows $(2 \,\hat{}\, x)/2 = (2 \,\hat{}\, (x - 1))$
  sorry

definition *max-signed-int* :: *'a::len word* **where**
  $max\text{-}signed\text{-}int = (2 \,\hat{}\, (LENGTH('a) - 1)) - 1$

definition *min-signed-int* :: *'a::len word* **where**
  $min\text{-}signed\text{-}int = -(2 \,\hat{}\, (LENGTH('a) - 1))$

definition *int-bottom* :: *'a::len word* × *'a word* **where**
  int-bottom = (*max-signed-int*, *min-signed-int*)

definition *int-top* :: *'a::len word* × *'a word* **where**
  int-top = (*min-signed-int*, *max-signed-int*)

lemma
  fixes *x* :: *'a::len word*
  shows $sint\ x \leq sint\ (((2 \,\hat{}\, (LENGTH('a) - 1)) - 1){::}'a\ word)$
  using *sint-greater-eq* **sorry**

value *sint* (*0::1 word*)
value *sint* (*1::1 word*)
value $sint\ (((2 \,\hat{}\, 0) - 1){::}1\ word)$

value $sint\ (((2 \,\hat{}\, 31) - 1){::}32\ word)$

lemma *max-signed*:
  fixes *a* :: *'a::len word*
  shows $sint\ a \leq sint\ (max\text{-}signed\text{-}int{::}'a\ word)$
proof (*cases sint a = sint* (*max-signed-int::'a word*))
  case *True*
  then show *?thesis*
    by *simp*
next
  case *False*
  have $sint\ a < sint\ (max\text{-}signed\text{-}int{::}'a\ word)$
    using *False* **unfolding** *max-signed-int-def* **sorry**
  then show *?thesis*

by *simp*
qed

lemma *min-signed*:
fixes *a* :: *'a::len word*
shows *sint a* ≥ *sint* (*min-signed-int::'a word*)
sorry

value *max-signed-int* :: *32 word*
value *int-bottom*::(*32 word* × *32 word*)
value *sint* (*2147483647::32 word*)
value *sint* (*2147483648::32 word*)

typedef (**overloaded**) (*'a::len*) *intstamp* =
{*bounds* :: (*'a word*, *'a word*) *prod* . ((*fst bounds*) ≤*s* (*snd bounds*) ∨ *bounds* = *int-bottom*)}
proof −
show *?thesis*
by *blast*
qed

setup-lifting *type-definition-intstamp*

lift-definition *lower* :: (*'a::len*) *intstamp* ⇒ *'a word*
is *prod.fst* ∘ *Rep-intstamp* **.**

lift-definition *upper* :: (*'a::len*) *intstamp* ⇒ *'a word*
is *prod.snd* ∘ *Rep-intstamp* **.**

lift-definition *lower-int* :: (*'a::len*) *intstamp* ⇒ *int*
is *sint* ∘ *prod.fst* **.**

lift-definition *upper-int* :: (*'a::len*) *intstamp* ⇒ *int*
is *sint* ∘ *prod.snd* **.**

lift-definition *range* :: (*'a::len*) *intstamp* ⇒ *int set*
is λ (*l*, *u*). {*sint l..sint u*} **.**

lift-definition *bounds* :: (*'a::len*) *intstamp* ⇒ (*'a word* × *'a word*)
is *Rep-intstamp* **.**

lift-definition *is-bottom* :: (*'a::len*) *intstamp* ⇒ *bool*
is λ *x*. *x* = *int-bottom* **.**

lift-definition *from-bounds* :: (*'a::len word* × *'a word*) ⇒ *'a intstamp*
is *Abs-intstamp* **.**

instantiation *intstamp* :: (*len*) *order*
begin

definition *less-eq-intstamp* :: *'a intstamp* ⇒ *'a intstamp* ⇒ *bool* **where**
less-eq-intstamp s1 s2 = (*range s1* ⊆ *range s2*)

definition *less-intstamp* :: *'a intstamp* ⇒ *'a intstamp* ⇒ *bool* **where**
less-intstamp s1 s2 = (*range s1* ⊂ *range s2*)

value *int-bottom*::(*1 word* × *1 word*)
value *sint* (*0*::*1 word*)
value *sint* (*1*::*1 word*)

value *int-bottom*::(*2 word* × *2 word*)
value *sint* (*1*::*2 word*)
value *sint* (*2*::*2 word*)
value *sint* ((*2* ^(*LENGTH*(*32*) − *1*) − *1*)::*32 word*) > *sint* ((− (*2* ^(*LENGTH*(*32*) − *1*)))::*32 word*)

lemma *bottom-is-bottom*:
assumes *is-bottom s*
shows *s* ≤ *a*
proof −
have *boundsdef*: *bounds s* = *int-bottom*
by (*metis assms bounds.transfer is-bottom.rep-eq*)
obtain *min max* **where** *bounds s* = (*max*, *min*)
by *fastforce*
then have *max* ≠ *min*
by (*metis boundsdef dual-order.eq-iff fst-conv int-bottom-def less-minus-one-simps*(*1*) *max-signed*
min-signed not-less sint-0 sint-n1 snd-conv)
then have *sint min* < *sint max*
by (*metis* ‹*bounds s* = (*max*, *min*)› *boundsdef max-signed boundsdef int-bottom-def signed-word-eqI*
order.not-eq-order-implies-strict prod.sel(*1*))
then have *range s* = {}
by (*simp add*: ‹*bounds s* = (*max*, *min*)› *bounds.transfer range-def*)
then show *?thesis*
by (*simp add*: *StampLattice.less-eq-intstamp-def*)
qed

lemma *bounds-has-value*:
fixes *x y* :: *int*
assumes *x* < *y*
shows *card* {*x*..*y*} > *0*
using *assms* **by** *simp*

lemma *bounds-has-no-value*:
fixes *x y* :: *int*
assumes *x* < *y*
shows *card* {*y*..*x*} = *0*

by (*simp add*: *assms*)

lemma *bottom-unique*:
fixes *a s* :: *'a intstamp*
assumes *is-bottom s*
shows $a \leq s \longleftrightarrow$ *is-bottom a*
proof –
have $\forall x.$ *sint* (*fst* (*bounds x*)) $\leq$ *sint* (*snd* (*bounds x*)) $\vee$ *is-bottom x*
using *Rep-intstamp* **by** (*auto simp add*: *word-sle-eq is-bottom-def bounds-def*)
then have $\forall x.$ (*card* (*range x*)) $> 0 \vee$ *is-bottom x*
by (*simp add*: *bounds.transfer case-prod-beta range-def*)
obtain *min max* **where** *boundsdef*: *bounds s* = (*max*, *min*)
by *fastforce*
have *nooverlap*: *sint min* < *sint max*
by (*metis assms bounds.transfer boundsdef fst-conv int-bottom-def is-bottom.rep-eq min-signed*
order.not-eq-order-implies-strict signed-word-eqI sint-0 snd-conv verit-la-disequality

zero-neq-one max-signed)
have *range s* = {*sint max..sint min*}
by (*simp add*: *bounds.transfer boundsdef range.rep-eq*)
then have *card* (*range s*) = *0*
by (*simp add*: *nooverlap*)
then have $\forall x.$ (*card* (*range x*)) $> 0 \longrightarrow s < x$
by (*auto simp add*: *less-intstamp-def* ‹*StampLattice.range s* = {*sint max..sint min*}›)
then show *?thesis*
by (*meson* ‹$\forall x.$ *0* < *card* (*StampLattice.range x*) $\vee$ *is-bottom x*› *bottom-is-bottom less-intstamp-def*
less-eq-intstamp-def leD)
qed

lemma *bottom-antisym*:
assumes *is-bottom x*
shows $x \leq y \Longrightarrow y \leq x \Longrightarrow x = y$
using *assms* **proof** (*cases is-bottom y*)
case *True*
then show *?thesis*
by (*metis Rep-intstamp-inverse assms is-bottom.rep-eq*)
next
case *False*
assume $y \leq x$
have $\neg(y \leq x)$
by (*simp add*: *assms False bottom-unique*)
then show *?thesis*
by (*simp add*: ‹$y \leq x$›)
qed

lemma *int-antisym*:

fixes *x y* :: *'a intstamp*
shows $x \leq y \Longrightarrow y \leq x \Longrightarrow x = y$
proof –
fix *x* :: *'a intstamp*
fix *y* :: *'a intstamp*
assume *xlessy*: $x \leq y$
assume *ylessx*: $y \leq x$
obtain *l1 u1* **where** *xdef*: *bounds x* = (*l1*, *u1*)
by *fastforce*
obtain *l2 u2* **where** *ydef*: *bounds y* = (*l2*, *u2*)
by *fastforce*
from *xlessy* **have** *s1*: {*sint l1..sint u1*} ⊆ {*sint l2..sint u2*} (**is** *?xlessy*)
using *xdef ydef less-eq-intstamp-def* **by** (*simp add*: *range-def bounds-def*)
from *ylessx* **have** *s2*: {*sint l2..sint u2*} ⊆ {*sint l1..sint u1*} (**is** *?ylessx*)
using *xdef ydef less-eq-intstamp-def* **by** (*simp add*: *range-def bounds-def*)
show $x = y$ **proof** (*cases is-bottom x*)
case *True*
then show *?thesis*
by (*simp add*: *ylessx xlessy bottom-antisym*)
next
case *False*
then show *?thesis*
sorry
qed
qed

instance
apply *standard*
using *less-eq-intstamp-def less-intstamp-def* **apply** (*simp*; *blast*)
by (*simp add*: *int-antisym less-eq-intstamp-def*)+
end

value *take-bit LENGTH*(*63*) *20*::*int*
value *take-bit LENGTH*(*63*) ((−*20*)::*int*)
value *bit* (*20*::*int64*) (*63*::*nat*)
value *bit* ((−*20*)::*int64*) (*63*::*nat*)

value ((−*20*)::*int64*) < (*20*::*int64*)

value *take-bit LENGTH*(*63*) ((−*20*)::*int*)

lift-definition *smax* :: *'a*::*len word* ⇒ *'a word* ⇒ *'a word*
is λ *a b*. (*if* (*sint a*) ≤ (*sint b*) *then b else a*) **.**

lift-definition *smin* :: *'a*::*len word* ⇒ *'a word* ⇒ *'a word*
is λ *a b*. (*if* (*sint a*) ≤ (*sint b*) *then a else b*) **.**

instantiation *intstamp* :: (*len*) *semilattice-inf*
begin

```
notation inf (infix ⊓ 65)

definition join-bounds :: 'a intstamp ⇒ 'a intstamp ⇒ ('a word × 'a word) where
  join-bounds s1 s2 = (smax (lower s1) (lower s2), smin (upper s1) (upper s2))

definition join-or-bottom :: 'a intstamp ⇒ 'a intstamp ⇒ ('a word × 'a word)
where
  join-or-bottom s1 s2 = (let bound = (join-bounds s1 s2) in
    if sint (fst bound) ≥ sint (snd bound) then int-bottom else bound)

definition inf-intstamp :: 'a intstamp ⇒ 'a intstamp ⇒ 'a intstamp where
  inf-intstamp s1 s2 = from-bounds (join-or-bottom s1 s2)

lemma always-valid:
  fixes s1 s2 :: 'a intstamp
  shows Rep-intstamp (from-bounds (join-or-bottom s1 s2)) = join-or-bottom s1 s2
 by (smt (z3) join-or-bottom-def from-bounds.transfer from-bounds-def mem-Collect-eq
word-sle-eq
      Abs-intstamp-inverse)

lemma invalid-join:
  fixes s1 s2 :: 'a intstamp
  assumes bound = join-bounds s1 s2
  assumes sint (fst bound) ≥ sint (snd bound)
  shows from-bounds int-bottom = s1 ⊓ s2
  using assms by (simp add: join-or-bottom-def inf-intstamp-def)

lemma unfold-bounds:
  bounds x = (lower x, upper x)
  by (simp add: bounds.transfer lower.rep-eq upper.rep-eq)

lemma int-inf-le1:
  fixes x y :: 'a intstamp
  shows (x ⊓ y) ≤ x
proof (cases is-bottom (x ⊓ y))
  case True
  then show ?thesis
    by (simp add: bottom-is-bottom)
next
  case False
  then show ?thesis
  using False proof −
  obtain l1 u1 where xdef: lower x = l1 ∧ upper x = u1
    by simp
  obtain l2 u2 where ydef: lower y = l2 ∧ upper y = u2
    by simp
  have joindef: x ⊓ y = from-bounds ((smax l1 l2, smin u1 u2))
    (is x ⊓ y = from-bounds (?l3, ?u3))
```

```
    by (smt (z3) StampLattice.inf-intstamp-def StampLattice.join-bounds-def al-
ways-valid False
        is-bottom.rep-eq join-or-bottom-def xdef ydef)
  have leq: {sint ?l3..sint ?u3} ⊆ {sint l1..sint u1}
    by (smt (z3) atLeastatMost-subset-iff smax.transfer smin.transfer)
  have (x ⊓ y) ≤ x = ({sint ?l3..sint ?u3} ⊆ {sint l1..sint u1})
    by (smt (z3) xdef less-eq-intstamp-def StampLattice.always-valid unfold-bounds
ydef range.rep-eq
      StampLattice.join-or-bottom-def bounds.abs-eq case-prod-conv inf-intstamp-def
False
        is-bottom.rep-eq join-bounds-def)
  then show (x ⊓ y) ≤ x
    using leq by simp
qed
qed

lemma int-inf-le2:
  fixes x y :: 'a intstamp
  shows (x ⊓ y) ≤ y
proof (cases is-bottom (x ⊓ y))
  case True
  then show ?thesis
    by (simp add: bottom-is-bottom)
next
  case False
  then show ?thesis
  using False proof −
  obtain l1 u1 where xdef: lower x = l1 ∧ upper x = u1
    by simp
  obtain l2 u2 where ydef: lower y = l2 ∧ upper y = u2
    by simp
  have joindef: x ⊓ y = from-bounds ((smax l1 l2, smin u1 u2))
    (is x ⊓ y = from-bounds (?l3, ?u3))
    by (smt (z3) False StampLattice.inf-intstamp-def StampLattice.join-bounds-def
always-valid ydef
        is-bottom.rep-eq join-or-bottom-def xdef)
  have leq: {sint ?l3..sint ?u3} ⊆ {sint l1..sint u1}
    by (smt (z3) atLeastatMost-subset-iff smax.transfer smin.transfer)
  have (x ⊓ y) ≤ y = ({sint ?l3..sint ?u3} ⊆ {sint l2..sint u2})
   by (smt (z3) less-eq-intstamp-def False StampLattice.always-valid unfold-bounds
range.rep-eq
      StampLattice.join-or-bottom-def bounds.abs-eq case-prod-conv inf-intstamp-def
xdef ydef
        is-bottom.rep-eq join-bounds-def)
  then show (x ⊓ y) ≤ y
    by (smt (z3) atLeastatMost-subset-iff smax.transfer smin.transfer)
qed
qed
```

lemma
assumes $x \leq y$
assumes *is-bottom y*
shows *is-bottom x*
using *assms* **by** (*auto simp add*: *bottom-unique bottom-is-bottom*)

lemma *int-inf-greatest*:
fixes *x y* :: *'a intstamp*
shows $x \leq y \Longrightarrow x \leq z \Longrightarrow x \leq y \sqcap z$
sorry

instance
apply *standard*
by (*simp add*: *local.int-inf-greatest local.int-inf-le2 local.int-inf-le1*)+

end

instantiation *intstamp* :: (*len*) *semilattice-sup*
begin

notation *sup* (**infix** ⊔ *65*)

instance apply *standard* **sorry**

end

instantiation *intstamp* :: (*len*) *bounded-lattice*
begin

notation *bot* (⊥ *50*)
notation *top* (⊤ *50*)

definition *bot-intstamp* = *int-bottom*
definition *top-intstamp* = *int-top*

instance apply *standard* **sorry**

end

value *sint* (*0*::*1 word*)
value *sint* (*1*::*1 word*)

datatype *Stamp* =
BottomStamp |
TopStamp |
VoidStamp |

Int8Stamp 8 intstamp |
Int16Stamp 16 intstamp |

Int32Stamp 32 intstamp |
Int64Stamp 64 intstamp

instantiation *Stamp* :: *order*
begin

fun *less-eq-Stamp* :: *Stamp* ⇒ *Stamp* ⇒ *bool* **where**
less-eq-Stamp BottomStamp - = True |
less-eq-Stamp - TopStamp = True |
less-eq-Stamp VoidStamp VoidStamp = True |
less-eq-Stamp (*Int8Stamp v1*) (*Int8Stamp v2*) = (*v1* ≤ *v2*) |
less-eq-Stamp (*Int16Stamp v1*) (*Int16Stamp v2*) = (*v1* ≤ *v2*) |
less-eq-Stamp (*Int32Stamp v1*) (*Int32Stamp v2*) = (*v1* ≤ *v2*) |
less-eq-Stamp (*Int64Stamp v1*) (*Int64Stamp v2*) = (*v1* ≤ *v2*) |
less-eq-Stamp - - = False

fun *less-Stamp* :: *Stamp* ⇒ *Stamp* ⇒ *bool* **where**
less-Stamp BottomStamp BottomStamp = False |
less-Stamp BottomStamp - = True |
less-Stamp TopStamp TopStamp = False |
less-Stamp - TopStamp = True |
less-Stamp VoidStamp VoidStamp = False |
less-Stamp (*Int8Stamp v1*) (*Int8Stamp v2*) = (*v1* < *v2*) |
less-Stamp (*Int16Stamp v1*) (*Int16Stamp v2*) = (*v1* < *v2*) |
less-Stamp (*Int32Stamp v1*) (*Int32Stamp v2*) = (*v1* < *v2*) |
less-Stamp (*Int64Stamp v1*) (*Int64Stamp v2*) = (*v1* < *v2*) |
less-Stamp - - = False

instance
apply *standard* **sorry**
end

instantiation *Stamp* :: *semilattice-inf*
begin

notation *inf* (**infix** ⊓ *65*)

fun *inf-Stamp* :: *Stamp* ⇒ *Stamp* ⇒ *Stamp* **where**
inf-Stamp BottomStamp - = BottomStamp |
inf-Stamp - BottomStamp = BottomStamp |
inf-Stamp TopStamp - = TopStamp |
inf-Stamp - TopStamp = TopStamp |
inf-Stamp VoidStamp VoidStamp = VoidStamp |
inf-Stamp (*Int8Stamp v1*) (*Int8Stamp v2*) = *Int8Stamp* (*v1* ⊓ *v2*) |
inf-Stamp (*Int16Stamp v1*) (*Int16Stamp v2*) = *Int16Stamp* (*v1* ⊓ *v2*) |
inf-Stamp (*Int32Stamp v1*) (*Int32Stamp v2*) = *Int32Stamp* (*v1* ⊓ *v2*) |
inf-Stamp (*Int64Stamp v1*) (*Int64Stamp v2*) = *Int64Stamp* (*v1* ⊓ *v2*)

instance

apply *standard* **sorry**
end

instantiation *Stamp* :: *semilattice-sup*
begin

notation *sup* (**infix** ⊔ *65*)

fun *sup-Stamp* :: *Stamp* ⇒ *Stamp* ⇒ *Stamp* **where**
sup-Stamp BottomStamp - = BottomStamp |
sup-Stamp - BottomStamp = BottomStamp |
sup-Stamp TopStamp - = TopStamp |
sup-Stamp - TopStamp = TopStamp |
sup-Stamp VoidStamp VoidStamp = VoidStamp |
sup-Stamp (*Int8Stamp v1*) (*Int8Stamp v2*) = *Int8Stamp* (*v1* ⊔ *v2*) |
sup-Stamp (*Int16Stamp v1*) (*Int16Stamp v2*) = *Int16Stamp* (*v1* ⊔ *v2*) |
sup-Stamp (*Int32Stamp v1*) (*Int32Stamp v2*) = *Int32Stamp* (*v1* ⊔ *v2*) |
sup-Stamp (*Int64Stamp v1*) (*Int64Stamp v2*) = *Int64Stamp* (*v1* ⊔ *v2*)

instance
apply *standard* **sorry**
end

instantiation *Stamp* :: *bounded-lattice*
begin

notation *bot* (⊥ *50*)
notation *top* (⊤ *50*)

definition *top-Stamp* :: *Stamp* **where**
top-Stamp = TopStamp
definition *bot-Stamp* :: *Stamp* **where**
bot-Stamp = BottomStamp

instance
apply *standard* **apply** (*simp add*: *bot-Stamp-def*)
by (*smt* (*verit*, *del-insts*) *less-eq-Stamp.simps*(*13*) *less-eq-Stamp.simps*(*2*) *sup.coboundedI1 sup-Stamp.simps*(*2*))

end

lemma [*code*]: *Rep-intstamp* (*from-bounds* (*l*, *u*)) = (*l*, *u*)
using *Abs-intstamp-inverse from-bounds.rep-eq*
sorry

code-datatype *Abs-intstamp*

end