

# GraalVM Stamp Theory

March 23, 2023

## 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

<b>1</b>	<b>Stamps: Type and Range Information</b>	<b>3</b>
1.1	Void Stamp . . . . .	3
1.2	Stamp Lattice . . . . .	4
1.2.1	Stamp Order . . . . .	5
1.2.2	Stamp Join . . . . .	6
1.2.3	Stamp Meet . . . . .	9
1.2.4	Stamp Bounds . . . . .	11
1.3	Java Stamp Methods . . . . .	13
1.4	Mapping to Values . . . . .	13
1.5	Generic Integer Stamp . . . . .	15

# 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  $\Rightarrow$  void  $\Rightarrow$  bool where
  less-eq-void a b = True
```

```
definition less-void :: void  $\Rightarrow$  void  $\Rightarrow$  bool where
  less-void a b = False
```

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

```
instantiation void :: semilattice-inf
begin
```

```
definition inf-void :: void  $\Rightarrow$  void  $\Rightarrow$  void where
  inf-void a b = VoidStamp
```

```
instance
  apply standard
    apply (simp add: less-eq-void-def)
    apply (simp add: less-eq-void-def)
    by (metis (mono-tags) void.exhaust)
end
```

```
instantiation void :: semilattice-sup
begin
```

**definition** *sup-void* :: *void*  $\Rightarrow$  *void*  $\Rightarrow$  *void* **where**  
*sup-void* *a b* = *VoidStamp*

**instance**  
**apply** *standard*  
**apply** (*simp add: less-eq-void-def*)  
**apply** (*simp add: less-eq-void-def*)  
**by** (*metis (mono-tags) void.exhaust*)

**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*  
**apply** (*simp add: less-eq-void-def*)  
**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*  $\Rightarrow$  *stamp*  $\Rightarrow$  *bool* **where**  
  *less-eq-stamp* (*intstamp* *l1* *u1*) (*intstamp* *l2* *u2*) = ( $\{l1..u1\} \subseteq \{l2..u2\}$ )

**fun** *less-stamp* :: *stamp*  $\Rightarrow$  *stamp*  $\Rightarrow$  *bool* **where**  
  *less-stamp* (*intstamp* *l1* *u1*) (*intstamp* *l2* *u2*) = ( $\{l1..u1\} \subset \{l2..u2\}$ )

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

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

**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*:  $x = \text{intstamp } l1 \ u1$   
    **using** *stamp.exhaust*  
    **by** *blast*  
  **obtain** *l2* *u2* **where** *ydef*:  $y = \text{intstamp } l2 \ u2$   
    **using** *stamp.exhaust*  
    **by** *blast*  
  **obtain** *l3* *u3* **where** *zdef*:  $z = \text{intstamp } l3 \ u3$   
    **using** *stamp.exhaust*  
    **by** *blast*  
  **have** *s1*:  $\{l1..u1\} \subseteq \{l2..u2\}$   
    **using**  $\langle x \leq y \rangle$  *less-eq-stamp.simps xdef ydef* **by** *blast*  
  **have** *s2*:  $\{l2..u2\} \subseteq \{l3..u3\}$   
    **using**  $\langle y \leq z \rangle$  *less-eq-stamp.simps ydef zdef* **by** *blast*  
  **from** *s1 s2* **have**  $\{l1..u1\} \subseteq \{l3..u3\}$   
    **by** (*meson dual-order.trans*)  
  **then show**  $x \leq z$

```

    using less-eq-stamp.simps
    using xdef zdef by presburger
qed

lemma antisym:
  fixes x y :: stamp
  shows  $x \leq y \implies y \leq x \implies x = y$ 
proof -
  fix x :: stamp
  fix y :: stamp
  assume xlessy:  $x \leq y$ 
  assume ylessx:  $y \leq x$ 
  obtain l1 u1 where xdef:  $x = \text{intstamp } l1 \ u1$ 
    using stamp.exhaust by blast
  obtain l2 u2 where ydef:  $y = \text{intstamp } l2 \ u2$ 
    using stamp.exhaust by blast

  from xlessy have s1:  $\{l1..u1\} \subseteq \{l2..u2\}$ 
    using less-eq-stamp.simps
    using xdef ydef by blast
  from ylessx have s2:  $\{l2..u2\} \subseteq \{l1..u1\}$ 
    using less-eq-stamp.simps
    using xdef ydef by blast
  have  $\{l1..u1\} \subseteq \{l2..u2\} \implies \{l2..u2\} \subseteq \{l1..u1\} \implies \{l1..u1\} = \{l2..u2\}$ 
    by fastforce
  then have s3:  $\{l1..u1\} = \{l2..u2\} \implies (l1 = l2) \wedge (u1 = u2)$ 

  sorry
  then have  $(l1 = l2) \wedge (u1 = u2) \implies x = y$ 
    using xdef ydef by fastforce
  then show  $x = y$ 
    using s1 s2 s3 by fastforce
qed

instance
  apply standard
  using less-le-not-le apply simp
  using order-refl apply simp
  using order-trans apply simp
  using antisym by simp
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  $\sqcap$  65)

fun inf-stamp :: stamp  $\Rightarrow$  stamp  $\Rightarrow$  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  $\sqcap$  y)  $\leq$  x
proof –
  fix x :: stamp
  fix y :: stamp
  obtain l1 u1 where xdef: x = intstamp l1 u1
    using stamp.exhaust by blast
  obtain l2 u2 where ydef: y = intstamp l2 u2
    using stamp.exhaust by blast
  have joindef: x  $\sqcap$  y = intstamp (max l1 l2) (min u1 u2)
    (is ?join = intstamp ?l3 ?u3)
    using inf-stamp.simps xdef ydef
    by force
  have leq: {?l3..?u3}  $\subseteq$  {l1..u1}
    by force
  have (x  $\sqcap$  y)  $\leq$  x = ({?l3..?u3}  $\subseteq$  {l1..u1})
    using xdef joindef inf-stamp.simps
    by force
  then show (x  $\sqcap$  y)  $\leq$  x
    using leq
    by fastforce
qed

lemma inf-le2:
  fixes x y :: stamp
  shows (x  $\sqcap$  y)  $\leq$  y
proof –
  fix x :: stamp
  fix y :: stamp
  obtain l1 u1 where xdef: x = intstamp l1 u1
    using stamp.exhaust by blast
  obtain l2 u2 where ydef: y = intstamp l2 u2
    using stamp.exhaust by blast
  have joindef: x  $\sqcap$  y = intstamp (max l1 l2) (min u1 u2)
    (is ?join = intstamp ?l3 ?u3)
    using inf-stamp.simps xdef ydef
    by force
  have leq: {?l3..?u3}  $\subseteq$  {l2..u2}
    by force
  have (x  $\sqcap$  y)  $\leq$  y = ({?l3..?u3}  $\subseteq$  {l2..u2})

```

```

    using ydef joindef
    by force
  then show  $(x \sqcap y) \leq y$ 
    using leq
    by fastforce
qed

```

```

lemma inf-greatest:
  fixes  $x\ y\ z :: \text{stamp}$ 
  shows  $x \leq y \implies x \leq z \implies x \leq (y \sqcap z)$ 
proof -
  fix  $x\ y\ z :: \text{stamp}$ 
  assume xlessy:  $x \leq y$ 
  assume xlessz:  $x \leq z$ 
  obtain  $l1\ u1$  where xdef:  $x = \text{intstamp } l1\ u1$ 
    using stamp.exhaust by blast
  obtain  $l2\ u2$  where ydef:  $y = \text{intstamp } l2\ u2$ 
    using stamp.exhaust by blast
  obtain  $l3\ u3$  where zdef:  $z = \text{intstamp } l3\ u3$ 
    using stamp.exhaust by blast
  obtain  $l4\ u4$  where yzdef:  $y \sqcap z = \text{intstamp } l4\ u4$ 
    by (meson inf-stamp.elims)
  have max4:  $l4 = \max\ l2\ l3$ 
    using yzdef ydef zdef inf-stamp.simps by simp
  have min4:  $u4 = \min\ u2\ u3$ 
    using yzdef ydef zdef inf-stamp.simps by simp
  have  $\{l1..u1\} \subseteq \{l2..u2\}$ 
    using xlessy xdef ydef
    using less-eq-stamp.simps by blast
  have  $\{l1..u1\} \subseteq \{l3..u3\}$ 
    using xlessz xdef zdef
    using less-eq-stamp.simps by blast
  have leq:  $\{l1..u1\} \subseteq \{l4..u4\}$ 
    using  $\langle \{l1..u1\} \subseteq \{l2..u2\} \rangle \langle \{l1..u1\} \subseteq \{l3..u3\} \rangle$  max4 min4 by auto
  have  $x \leq (y \sqcap z) = (\{l1..u1\} \subseteq \{l4..u4\})$ 
    by (simp add: xdef yzdef)
  then show  $x \leq (y \sqcap z)$ 
    using leq
    by fastforce
qed

```

```

instance
  apply standard
  using inf-le1 apply simp
  using inf-le2 apply simp
  using inf-greatest by simp
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**  $\sqcup$  65)

**fun** *sup-stamp* :: *stamp*  $\Rightarrow$  *stamp*  $\Rightarrow$  *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**  $x \leq x \sqcup y$

**proof** –

**fix** *x* :: *stamp*

**fix** *y* :: *stamp*

**obtain** *l1 u1* **where** *xdef*:  $x = \text{intstamp } l1 \ u1$

**using** *stamp.exhaust* **by** *blast*

**obtain** *l2 u2* **where** *ydef*:  $y = \text{intstamp } l2 \ u2$

**using** *stamp.exhaust* **by** *blast*

**have** *joindef*:  $x \sqcup y = \text{intstamp } (\text{min } l1 \ l2) \ (\text{max } u1 \ u2)$

(**is** *?join* = *intstamp* *?l3* *?u3*)

**using** *inf-stamp.simps* *xdef* *ydef*

**by** *force*

**have** *leq*:  $\{l1..u1\} \subseteq \{?l3..?u3\}$

**by** *simp*

**have**  $x \leq x \sqcup y = (\{l1..u1\} \subseteq \{?l3..?u3\})$

**using** *xdef* *joindef* *inf-stamp.simps*

**by** *force*

**then show**  $x \leq x \sqcup y$

**using** *leq*

**by** *fastforce*

**qed**

**lemma** *sup-ge2*:

**fixes** *x y* :: *stamp*

**shows**  $y \leq x \sqcup y$

**proof** –

**fix** *x* :: *stamp*

**fix** *y* :: *stamp*

**obtain** *l1 u1* **where** *xdef*:  $x = \text{intstamp } l1 \ u1$

**using** *stamp.exhaust* **by** *blast*

**obtain** *l2 u2* **where** *ydef*:  $y = \text{intstamp } l2 \ u2$

**using** *stamp.exhaust* **by** *blast*

**have** *joindef*:  $x \sqcup y = \text{intstamp } (\text{min } l1 \ l2) \ (\text{max } u1 \ u2)$

(**is** *?join* = *intstamp* *?l3* *?u3*)

```

    using inf-stamp.simps xdef ydef
  by force
have leq:  $\{l2..u2\} \subseteq \{?l3..?u3\}$  (is ?subset-thesis)
  by simp
have ?thesis = (?subset-thesis)
  using ydef joindef sup-stamp.simps less-eq-stamp.simps
  by (metis StampLattice.sup-ge1 max.commute min.commute sup-stamp.elims)
then show ?thesis
  using leq
  by fastforce
qed

```

lemma sup-least:

```

  fixes x y z :: stamp
  shows  $y \leq x \implies z \leq x \implies ((y \sqcup z) \leq x)$ 
proof -
  fix x y z :: stamp
  assume xlessy:  $y \leq x$ 
  assume xlessz:  $z \leq x$ 
  obtain l1 u1 where xdef:  $x = \text{intstamp } l1 \ u1$ 
    using stamp.exhaust by blast
  obtain l2 u2 where ydef:  $y = \text{intstamp } l2 \ u2$ 
    using stamp.exhaust by blast
  obtain l3 u3 where zdef:  $z = \text{intstamp } l3 \ u3$ 
    using stamp.exhaust by blast
  have yzdef:  $y \sqcup z = \text{intstamp } (\min l2 \ l3) \ (\max u2 \ u3)$ 
    (is ?meet =  $\text{intstamp } ?l4 \ ?u4$ )
    using sup-stamp.simps
    by (simp add: ydef zdef)
  have s1:  $\{l2..u2\} \subseteq \{l1..u1\}$ 
    using xlessy xdef ydef
    using less-eq-stamp.simps by blast
  have s2:  $\{l3..u3\} \subseteq \{l1..u1\}$ 
    using xlessz xdef zdef
    using less-eq-stamp.simps by blast
  have leq:  $\{?l4..?u4\} \subseteq \{l1..u1\}$  (is ?subset-thesis)
    using s1 s2 unfolding atLeastatMost-subset-iff

  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 xlessz ydef
    zdef)
  have  $(y \sqcup z \leq x) = ?subset-thesis$ 
    using yzdef xdef less-eq-stamp.simps
    by simp
  then show  $(y \sqcup z \leq x)$ 
    using leq by fastforce
qed

```

instance

```

apply standard
using sup-ge1 apply simp
using sup-ge2 apply simp
using sup-least by simp
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 ( $\perp$  50)
notation top ( $\top$  50)

```

```

definition width-min :: nat  $\Rightarrow$  int64 where
  width-min bits =  $-(2^{\wedge}(\text{bits}-1))$ 

```

```

definition width-max :: nat  $\Rightarrow$  int64 where
  width-max bits =  $(2^{\wedge}(\text{bits}-1)) - 1$ 

```

```

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
  using assms unfolding width-min-def width-max-def by simp

```

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  $(\perp) \leq a$

**proof** –

obtain *min max* where *bot-def*: $\perp$  = *intstamp* *max min*

using *bot-stamp-def*

by *force*

have *min* < *max*

using *bot-def*

unfolding *bot-stamp-def width-min-def width-max-def*

using *word-gt-0* by *fastforce*

then have {*max..min*} = {}

using *bot-def*

unfolding *bot-stamp-def width-min-def width-max-def*

by *auto*

then show ?*thesis*

unfolding *bot-stamp-def*

using *less-eq-stamp.simps*

by (*simp add: stamp.induct*)

qed

**lemma** *top-greatest*:

fixes *a* :: *stamp*

shows  $a \leq (\top)$

**proof** –

obtain *min max* where *top-def*: $\top$  = *intstamp* *min max*

using *top-stamp-def*

by *force*

have *max-is-max*:  $\neg(\exists n. n > \text{max})$

by (*metis stamp.inject top-def top-stamp-def word-order.extremum-strict*)

have *min-is-min*:  $\neg(\exists n. n < \text{min})$

by (*metis not-less-iff-gr-or-eq stamp.inject top-def top-stamp-def word-coorder.not-eq-extremum*)

have  $\neg(\exists l\ u. \{\text{min..max}\} < \{l..u\})$

using *max-is-max min-is-min*

by (*metis atLeastatMost-psubset-iff not-less*)

then show ?*thesis*

unfolding *top-stamp-def*

using *less-eq-stamp.simps*

using *less-eq-stamp.elims(3)* by *fastforce*

qed

**instance**

apply *standard*

using *bot-least* apply *simp*

```

    using top-greatest by simp
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  $\Rightarrow$  bool **where**  
*is-unrestricted*  $s = (\top = s)$

**fun** *is-empty* :: stamp  $\Rightarrow$  bool **where**  
*is-empty*  $s = (\perp = s)$

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

**definition** *always-distinct* :: stamp  $\Rightarrow$  stamp  $\Rightarrow$  bool **where**  
*always-distinct* stamp1 stamp2 = ( $\perp = (stamp1 \sqcap stamp2)$ )

**definition** *never-distinct* :: stamp  $\Rightarrow$  stamp  $\Rightarrow$  bool **where**  
*never-distinct* stamp1 stamp2 =  
 (*as-constant* stamp1 = *as-constant* stamp2  $\wedge$  *as-constant* stamp1  $\neq$  None)

### 1.4 Mapping to Values

**fun** *valid-value* :: stamp  $\Rightarrow$  Value  $\Rightarrow$  bool **where**  
*valid-value* (intstamp  $l\ u$ ) (IntVal  $b\ v$ ) = ( $v \in \{l..u\}$ ) |  
*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*:  
 $\neg(\text{valid-value } (\perp) v)$   
**unfolding** *bot-stamp-def*  
**using** *valid-value.elims(2)* **by** *fastforce*

**lemma** *join-values*:  
**assumes** *joined* =  $x\text{-stamp} \sqcap y\text{-stamp}$   
**shows** *valid-value* *joined*  $x \iff (\text{valid-value } x\text{-stamp } x \wedge \text{valid-value } y\text{-stamp } x)$   
**proof** (cases  $x$ )  
**case** *UndefVal*  
**then show** *?thesis*  
**using** *valid-value.elims(2)* **by** *blast*

```

next
  case (IntVal b x3)
  obtain lx ux where xdef: x-stamp = intstamp lx ux
    using stamp.exhaust by blast
  obtain ly uy where ydef: y-stamp = intstamp ly uy
    using stamp.exhaust by blast
  obtain v where x = IntVal b v
    using IntVal by blast
  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⟩ xdef ydef by force
next
  case (ObjRef x5)
  then show ?thesis
    using valid-value.elims(2) by blast
next
  case (ObjStr x6)
  then show ?thesis
    using valid-value.elims(2) by blast
qed

```

```

lemma disjoint-empty:
  fixes x-stamp y-stamp :: stamp
  assumes  $\perp = x\text{-stamp} \sqcap y\text{-stamp}$ 
  shows  $\neg(\text{valid-value } x\text{-stamp } x \wedge \text{valid-value } y\text{-stamp } x)$ 
  using assms bottom-range-empty join-values
  by blast

```

## 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  $\{l1..u1\} \subseteq \{l2..u2\} = \text{less-eq-alt } (l1, u1) (l2, u2)$ 
  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  $(x::\text{int}) > 0$ 
  shows  $(2^x)/2 = (2^{(x-1)})$ 
  sorry

```

```

definition max-signed-int :: 'a::len word where
  max-signed-int =  $(2^{(LENGTH('a) - 1)} - 1)$ 

```

```

definition min-signed-int :: 'a::len word where
  min-signed-int =  $-(2^{(LENGTH('a) - 1)})$ 

```

```

definition int-bottom :: 'a::len word  $\times$  'a word where
  int-bottom = (max-signed-int, min-signed-int)

```

```

definition int-top :: 'a::len word  $\times$  'a word where
  int-top = (min-signed-int, max-signed-int)

```

```

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

```

```

value sint (0::1 word)
value sint (1::1 word)
value sint  $((2^0 - 1)::1 word)$ 

```

```

value sint  $((2^{31} - 1)::32 word)$ 

```

```

lemma max-signed:
  fixes  $a :: 'a::len word$ 
  shows  $\text{sint } a \leq \text{sint } (\text{max-signed-int}::'a word)$ 
proof (cases  $\text{sint } a = \text{sint } (\text{max-signed-int}::'a word)$ )
  case True
  then show ?thesis by simp
next
  case False
  have  $\text{sint } a < \text{sint } (\text{max-signed-int}::'a word)$ 
  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 (smt (z3) mem-Collect-eq prod.sel(1) prod.sel(2) signed-minus-1 sint-0)
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*  $\Rightarrow$  'a *intstamp*  $\Rightarrow$  bool **where**  
*less-eq-intstamp* *s1* *s2* = (*range* *s1*  $\subseteq$  *range* *s2*)

**definition** *less-intstamp* :: 'a *intstamp*  $\Rightarrow$  'a *intstamp*  $\Rightarrow$  bool **where**  
*less-intstamp* *s1* *s2* = (*range* *s1*  $\subset$  *range* *s2*)

**value** *int-bottom*::(1 word  $\times$  1 word)  
**value** *sint* (0::1 word)  
**value** *sint* (1::1 word)

**value** *int-bottom*::(2 word  $\times$  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*  $\leq$  *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*  $\neq$  *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*  
**unfolding** *boundsdef* *int-bottom-def*  
**using** *max-signed*  
**by** (*metis*  $\langle$ *bounds* *s* = (*max*, *min*) $\rangle$  *boundsdef* *int-bottom-def* *order.not-eq-order-implies-strict*  
*prod.sel*(1) *signed-word-eqI*)  
**then have** *range* *s* = {}  
**unfolding** *range-def* *bounds-def*  
**by** (*simp* *add*:  $\langle$ *bounds* *s* = (*max*, *min*) $\rangle$  *bounds.transfer*)  
**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** *auto*

```

lemma bounds-has-no-value:
  fixes  $x\ y :: \text{int}$ 
  assumes  $x < y$ 
  shows  $\text{card } \{y..x\} = 0$ 
  using assms by auto

lemma bottom-unique:
  fixes  $a\ s :: 'a\ \text{intstamp}$ 
  assumes is-bottom  $s$ 
  shows  $a \leq s \longleftrightarrow \text{is-bottom } a$ 
proof -
  have  $\forall x. \text{sint } (\text{fst } (\text{bounds } x)) \leq \text{sint } (\text{snd } (\text{bounds } x)) \vee \text{is-bottom } x$ 
    unfolding bounds-def is-bottom-def
    using Rep-intstamp
    using word-sle-eq by auto
  then have  $\forall x. (\text{card } (\text{range } x)) > 0 \vee \text{is-bottom } x$ 
    unfolding range-def using bounds-has-value
    by (simp add: bounds.transfer case-prod-beta)
  obtain  $\text{min } \text{max}$  where boundsdef:  $\text{bounds } s = (\text{max}, \text{min})$ 
    by fastforce
  have nooverlap:  $\text{sint } \text{min} < \text{sint } \text{max}$ 
    using max-signed
  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)
  have  $\text{range } s = \{\text{sint } \text{max}.. \text{sint } \text{min}\}$ 
    by (simp add: bounds.transfer boundsdef range.rep-eq)
  then have  $\text{card } (\text{range } s) = 0$ 
    using nooverlap bounds-has-no-value by simp
  then have  $\forall x. (\text{card } (\text{range } x)) > 0 \longrightarrow s < x$ 
    using  $\langle \text{StampLattice.range } s = \{\text{sint } \text{max}.. \text{sint } \text{min}\} \rangle$  atLeastatMost-empty
less-intstamp-def by auto
  then show ?thesis
    by (meson  $\langle \forall x. 0 < \text{card } (\text{StampLattice.range } x) \vee \text{is-bottom } x \rangle$  bottom-is-bottom
leD less-eq-intstamp-def less-intstamp-def)
qed

lemma bottom-antisym:
  assumes is-bottom  $x$ 
  shows  $x \leq y \implies y \leq x \implies x = y$ 
  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  $\neg(y \leq x)$ 
  using bottom-unique False assms
  by simp
then show ?thesis
  using  $\langle y \leq x \rangle$  by auto
qed

lemma int-antisym:
  fixes  $x\ y :: 'a\ intstamp$ 
  shows  $x \leq y \implies y \leq x \implies x = y$ 
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  $s1: \{sint\ l1..sint\ u1\} \subseteq \{sint\ l2..sint\ u2\}$  (is ?xlessy)
    using xdef ydef unfolding bounds-def range-def less-eq-intstamp-def
    by simp
  from ylessx have  $s2: \{sint\ l2..sint\ u2\} \subseteq \{sint\ l1..sint\ u1\}$  (is ?ylessx)
    using xdef ydef unfolding bounds-def range-def less-eq-intstamp-def
    by simp
  show  $x = y$  proof (cases is-bottom x)
    case True
    then show ?thesis using bottom-antisym xlessy ylessx
      by simp
    next
    case False
    then show ?thesis sorry
  qed
qed

instance
  apply standard
  apply (simp add: less-eq-intstamp-def less-intstamp-def less-le-not-le)
  apply blast
  using less-eq-intstamp-def apply force
  using less-eq-intstamp-def apply force
  by (simp add: int-antisym)
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**  $((-20)::\text{int64}) < (20::\text{int64})$

**value**  $\text{take-bit LENGTH}(63) ((-20)::\text{int})$

**lift-definition**  $\text{smax} :: 'a::\text{len word} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word}$   
**is**  $\lambda a b. (\text{if } (\text{sint } a) \leq (\text{sint } b) \text{ then } b \text{ else } a) .$

**lift-definition**  $\text{smin} :: 'a::\text{len word} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word}$   
**is**  $\lambda a b. (\text{if } (\text{sint } a) \leq (\text{sint } b) \text{ then } a \text{ else } b) .$

**instantiation**  $\text{intstamp} :: (\text{len}) \text{ semilattice-inf}$   
**begin**

**notation**  $\text{inf}$  (**infix**  $\sqcap$  65)

**definition**  $\text{join-bounds} :: 'a \text{ intstamp} \Rightarrow 'a \text{ intstamp} \Rightarrow ('a \text{ word} \times 'a \text{ word})$  **where**  
 $\text{join-bounds } s1 \ s2 = (\text{smax } (\text{lower } s1) (\text{lower } s2), \text{smin } (\text{upper } s1) (\text{upper } s2))$

**definition**  $\text{join-or-bottom} :: 'a \text{ intstamp} \Rightarrow 'a \text{ intstamp} \Rightarrow ('a \text{ word} \times 'a \text{ word})$   
**where**  
 $\text{join-or-bottom } s1 \ s2 = (\text{let bound} = (\text{join-bounds } s1 \ s2) \text{ in}$   
 $\text{if } \text{sint } (\text{fst bound}) \geq \text{sint } (\text{snd bound}) \text{ then } \text{int-bottom} \text{ else bound})$

**definition**  $\text{inf-intstamp} :: 'a \text{ intstamp} \Rightarrow 'a \text{ intstamp} \Rightarrow 'a \text{ intstamp}$  **where**  
 $\text{inf-intstamp } s1 \ s2 = \text{from-bounds } (\text{join-or-bottom } s1 \ s2)$

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

**lemma** *invalid-join*:  
**fixes**  $s1 \ s2 :: 'a \text{ intstamp}$   
**assumes**  $\text{bound} = \text{join-bounds } s1 \ s2$   
**assumes**  $\text{sint } (\text{fst bound}) \geq \text{sint } (\text{snd bound})$   
**shows**  $\text{from-bounds int-bottom} = s1 \sqcap s2$   
**using**  $\text{assms}(1) \text{ assms}(2) \text{ inf-intstamp-def join-or-bottom-def}$  **by** *presburger*

**lemma** *unfold-bounds*:  
 $\text{bounds } x = (\text{lower } x, \text{upper } x)$   
**by**  $(\text{simp add: bounds.transfer lower.rep-eq upper.rep-eq})$

**lemma** *int-inf-le1*:  
**fixes**  $x \ y :: 'a \text{ intstamp}$   
**shows**  $(x \sqcap y) \leq x$

```

proof (cases is-bottom (x  $\sqcap$  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  $\wedge$  upper x = u1
    by fastforce
  obtain l2 u2 where ydef: lower y = l2  $\wedge$  upper y = u2
    by fastforce
  have joindef: x  $\sqcap$  y = from-bounds ((smax l1 l2, smin u1 u2))
    (is x  $\sqcap$  y = from-bounds (?l3, ?u3))
  using False
    by (smt (z3) StampLattice.inf-intstamp-def StampLattice.join-bounds-def al-
ways-valid is-bottom.rep-eq join-or-bottom-def xdef ydef)
  have leq: {sint ?l3..sint ?u3}  $\subseteq$  {sint l1..sint u1}
    by (smt (z3) atLeastatMost-subset-iff smax.transfer smin.transfer)
  have (x  $\sqcap$  y)  $\leq$  x = ({sint ?l3..sint ?u3}  $\subseteq$  {sint l1..sint u1})
    using xdef joindef range-def less-eq-intstamp-def
    by (smt (z3) False StampLattice.always-valid StampLattice.join-or-bottom-def
bounds.abs-eq case-prod-conv inf-intstamp-def is-bottom.rep-eq join-bounds-def range.rep-eq
unfold-bounds ydef)
  then show (x  $\sqcap$  y)  $\leq$  x
    using leq
    by fastforce
qed
qed

lemma int-inf-le2:
  fixes x y :: 'a intstamp
  shows (x  $\sqcap$  y)  $\leq$  y
proof (cases is-bottom (x  $\sqcap$  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  $\wedge$  upper x = u1
    by fastforce
  obtain l2 u2 where ydef: lower y = l2  $\wedge$  upper y = u2
    by fastforce
  have joindef: x  $\sqcap$  y = from-bounds ((smax l1 l2, smin u1 u2))
    (is x  $\sqcap$  y = from-bounds (?l3, ?u3))
  using False
    by (smt (z3) StampLattice.inf-intstamp-def StampLattice.join-bounds-def al-

```

```

ways-valid is-bottom.rep-eq join-or-bottom-def xdef ydef)
  have leq: {sint ?l3..sint ?u3}  $\subseteq$  {sint l1..sint u1}
    by (smt (z3) atLeastatMost-subset-iff smax.transfer smin.transfer)
  have (x  $\sqcap$  y)  $\leq$  y = ({sint ?l3..sint ?u3}  $\subseteq$  {sint l2..sint u2})
    using xdef joindef range-def less-eq-intstamp-def
    by (smt (z3) False StampLattice.always-valid StampLattice.join-or-bottom-def
        bounds.abs-eq case-prod-conv inf-intstamp-def is-bottom.rep-eq join-bounds-def range.rep-eq
        unfold-bounds ydef)
  then show (x  $\sqcap$  y)  $\leq$  y
    using leq
    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 bottom-is-bottom assms
  using bottom-unique by auto

```

```

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

```

```

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

```

end

```

instantiation intstamp :: (len) semilattice-sup
begin

```

```

notation sup (infix  $\sqcup$  65)

```

```

instance sorry

```

end

```

instantiation intstamp :: (len) bounded-lattice
begin

```

```

notation bot ( $\perp$  50)
notation top ( $\top$  50)

```

```

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

instance 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  $\sqcap$  65)

```

```

fun inf-Stamp :: Stamp  $\Rightarrow$  Stamp  $\Rightarrow$  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  $\sqcap$  v2) |
  inf-Stamp (Int16Stamp v1) (Int16Stamp v2) = Int16Stamp (v1  $\sqcap$  v2) |
  inf-Stamp (Int32Stamp v1) (Int32Stamp v2) = Int32Stamp (v1  $\sqcap$  v2) |
  inf-Stamp (Int64Stamp v1) (Int64Stamp v2) = Int64Stamp (v1  $\sqcap$  v2)

```

```

instance
  apply standard sorry
end

```

```

instantiation Stamp :: semilattice-sup
begin

```

```

notation sup (infix  $\sqcup$  65)

```

```

fun sup-Stamp :: Stamp  $\Rightarrow$  Stamp  $\Rightarrow$  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  $\sqcup$  v2) |
  sup-Stamp (Int16Stamp v1) (Int16Stamp v2) = Int16Stamp (v1  $\sqcup$  v2) |
  sup-Stamp (Int32Stamp v1) (Int32Stamp v2) = Int32Stamp (v1  $\sqcup$  v2) |
  sup-Stamp (Int64Stamp v1) (Int64Stamp v2) = Int64Stamp (v1  $\sqcup$  v2)

```

```

instance
  apply standard sorry
end

```

```

instantiation Stamp :: bounded-lattice
begin

```

```

notation bot ( $\perp$  50)
notation top ( $\top$  50)

```



```

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

instance
  apply standard sorry
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

```