Interval Arithmetic on 32-bit Words

Rose Bohrer

February 6, 2026

Abstract

This article implements conservative interval arithmetic computations, then uses this interval arithmetic to implement a simple programming language where all terms have 32-bit signed word values, with explicit infinities for terms outside the representable bounds. Our target use case is interpreters for languages that must have a well-understood low-level behavior.

We include a formalization of bounded-length strings which are used for the identifiers of our language. Bounded-length identifiers are useful in some applications, for example the Differential_Dynamic_Logic [1] article, where a Euclidean space indexed by identifiers demands that identifiers are finitely many.

Contents

1 Interval arithmetic definitions **7**
1.1 Syntax . . . 7

2 Preliminary lemmas **9**
2.1 Case analysis lemmas . . . 9
2.2 Trivial arithmetic lemmas . . . 13

3 Arithmetic operations **14**
3.1 Addition upper bound . . . 14
3.2 Addition lower bound . . . 28
3.3 Max function . . . 44
3.4 Multiplication upper bound . . . 48
3.5 Exact multiplication . . . 50
3.6 Multiplication upper bound . . . 64
3.7 Minimum function . . . 72
3.8 Multiplication lower bound . . . 76
3.9 Negation . . . 90
3.10 Comparison . . . 91
3.11 Absolute value . . . 92

4 Finite Strings **93**

5 Syntax **104**

6 Semantics **105**

7 Soundness proofs **108**

theory *Interval-Word32*
imports
Complex-Main
Word-Lib.Word-Lib-Sumo
begin

abbreviation *signed-real-of-word* :: ‹*'a*::*len word* ⇒ *real*›
where ‹*signed-real-of-word* ≡ *signed*›

lemma (**in** *linordered-idom*) *signed-less-numeral-iff*:
‹*signed w* < *numeral n* ⟷ *sint w* < *numeral n*› (**is** ‹*?P* ⟷ *?Q*›)
proof −
have ‹*?Q* ⟷ *of-int* (*sint w*) < *of-int* (*numeral n*)›
by (*simp only*: *of-int-less-iff*)
also have ‹... ⟷ *?P*›
by (*transfer fixing*: *less less-eq n*) *simp*
finally show *?thesis* **..**
qed

lemma (**in** *linordered-idom*) *signed-less-neg-numeral-iff*:
‹*signed w* < − *numeral n* ⟷ *sint w* < − *numeral n*› (**is** ‹*?P* ⟷ *?Q*›)
proof −
have ‹*?Q* ⟷ *of-int* (*sint w*) < *of-int* (− *numeral n*)›
by (*simp only*: *of-int-less-iff*)
also have ‹... ⟷ *?P*›
by (*transfer fixing*: *less less-eq uminus n*) *simp*
finally show *?thesis* **..**
qed

lemma (**in** *linordered-idom*) *numeral-less-signed-iff*:
‹*numeral n* < *signed w* ⟷ *numeral n* < *sint w*› (**is** ‹*?P* ⟷ *?Q*›)
proof −
have ‹*?Q* ⟷ *of-int* (*numeral n*) < *of-int* (*sint w*)›
by (*simp only*: *of-int-less-iff*)
also have ‹... ⟷ *?P*›
by (*transfer fixing*: *less less-eq n*) *simp*
finally show *?thesis* **..**
qed

lemma (**in** *linordered-idom*) *neg-numeral-less-signed-iff*:
‹− *numeral n* < *signed w* ⟷ − *numeral n* < *sint w*› (**is** ‹*?P* ⟷ *?Q*›)
proof −
have ‹*?Q* ⟷ *of-int* (− *numeral n*) < *of-int* (*sint w*)›
by (*simp only*: *of-int-less-iff*)
also have ‹... ⟷ *?P*›
by (*transfer fixing*: *less less-eq uminus n*) *simp*
finally show *?thesis* **..**
qed

lemma (**in** *linordered-idom*) *signed-nonnegative-iff*:
‹*0* ≤ *signed w* ⟷ *0* ≤ *sint w*› (**is** ‹*?P* ⟷ *?Q*›)
proof −
have ‹*?Q* ⟷ *of-int 0* ≤ *of-int* (*sint w*)›
by (*simp only*: *of-int-le-iff*)
also have ‹... ⟷ *?P*›
by (*transfer fixing*: *less-eq*) *simp*
finally show *?thesis* **..**
qed

lemma *signed-real-of-word-plus-numeral-eq-signed-real-of-word-iff*:
‹*signed-real-of-word v* + *numeral n* = *signed-real-of-word w*
⟷ *sint v* + *numeral n* = *sint w*› (**is** ‹*?P* ⟷ *?Q*›)
proof −
have ‹*?Q* ⟷ *real-of-int* (*sint v* + *numeral n*) = *real-of-int* (*sint w*)›
by (*simp only*: *of-int-eq-iff*)
also have ‹... ⟷ *?P*›
by *simp*
finally show *?thesis* **..**
qed

lemma *signed-real-of-word-sum-less-eq-numeral-iff*:
‹*signed-real-of-word v* + *signed-real-of-word w* ≤ *numeral n*
⟷ *sint v* + *sint w* ≤ *numeral n*› (**is** ‹*?P* ⟷ *?Q*›)
proof −
have ‹*?Q* ⟷ *real-of-int* (*sint v* + *sint w*) ≤ *real-of-int* (*numeral n*)›
by (*simp only*: *of-int-le-iff*)
also have ‹... ⟷ *?P*›
by *simp*
finally show *?thesis* **..**
qed

lemma *signed-real-of-word-sum-less-eq-neg-numeral-iff*:
‹*signed-real-of-word v* + *signed-real-of-word w* ≤ − *numeral n*
⟷ *sint v* + *sint w* ≤ − *numeral n*› (**is** ‹*?P* ⟷ *?Q*›)
proof −
have ‹*?Q* ⟷ *real-of-int* (*sint v* + *sint w*) ≤ *real-of-int* (− *numeral n*)›
by (*simp only*: *of-int-le-iff*)
also have ‹... ⟷ *?P*›
by *simp*
finally show *?thesis* **..**
qed

lemma *signed-real-of-word-sum-less-numeral-iff*:
‹*signed-real-of-word v* + *signed-real-of-word w* < *numeral n*
⟷ *sint v* + *sint w* < *numeral n*› (**is** ‹*?P* ⟷ *?Q*›)
proof −
have ‹*?Q* ⟷ *real-of-int* (*sint v* + *sint w*) < *real-of-int* (*numeral n*)›

```
    by (simp only: of-int-less-iff)
  also have ‹... ⟷ ?P›
    by simp
  finally show ?thesis ..
qed

lemma signed-real-of-word-sum-less-neg-numeral-iff:
  ‹signed-real-of-word v + signed-real-of-word w < - numeral n
    ⟷ sint v + sint w < - numeral n› (is ‹?P ⟷ ?Q›)
proof -
  have ‹?Q ⟷ real-of-int (sint v + sint w) < real-of-int (- numeral n)›
    by (simp only: of-int-less-iff)
  also have ‹... ⟷ ?P›
    by simp
  finally show ?thesis ..
qed

lemma numeral-less-eq-signed-real-of-word-sum:
  ‹numeral n ≤ signed-real-of-word v + signed-real-of-word w
    ⟷ numeral n ≤ sint v + sint w› (is ‹?P ⟷ ?Q›)
proof -
  have ‹?Q ⟷ real-of-int (numeral n) ≤ real-of-int (sint v + sint w)›
    by (simp only: of-int-le-iff)
  also have ‹... ⟷ ?P›
    by simp
  finally show ?thesis ..
qed

lemma neg-numeral-less-eq-signed-real-of-word-sum:
  ‹- numeral n ≤ signed-real-of-word v + signed-real-of-word w
    ⟷ - numeral n ≤ sint v + sint w› (is ‹?P ⟷ ?Q›)
proof -
  have ‹?Q ⟷ real-of-int (- numeral n) ≤ real-of-int (sint v + sint w)›
    by (simp only: of-int-le-iff)
  also have ‹... ⟷ ?P›
    by simp
  finally show ?thesis ..
qed

lemma numeral-less-signed-real-of-word-sum:
  ‹numeral n < signed-real-of-word v + signed-real-of-word w
    ⟷ numeral n < sint v + sint w› (is ‹?P ⟷ ?Q›)
proof -
  have ‹?Q ⟷ real-of-int (numeral n) < real-of-int (sint v + sint w)›
    by (simp only: of-int-less-iff)
  also have ‹... ⟷ ?P›
    by simp
  finally show ?thesis ..
qed
```

lemma *neg-numeral-less-signed-real-of-word-sum*:
‹− *numeral n* < *signed-real-of-word v* + *signed-real-of-word w*
⟷ − *numeral n* < *sint v* + *sint w*› (**is** ‹*?P* ⟷ *?Q*›)
proof −
have ‹*?Q* ⟷ *real-of-int* (− *numeral n*) < *real-of-int* (*sint v* + *sint w*)›
by (*simp only*: *of-int-less-iff*)
also have ‹... ⟷ *?P*›
by *simp*
finally show *?thesis* **..**
qed

lemmas *real-of-word-simps* [*simp*] = *signed-less-numeral-iff* [**where** *?'a* = *real*]
numeral-less-signed-iff [**where** *?'a* = *real*]
signed-less-neg-numeral-iff [**where** *?'a* = *real*]
neg-numeral-less-signed-iff [**where** *?'a* = *real*]
signed-nonnegative-iff [**where** *?'a* = *real*]

lemmas *more-real-of-word-simps* =
signed-real-of-word-plus-numeral-eq-signed-real-of-word-iff
signed-real-of-word-sum-less-eq-numeral-iff
signed-real-of-word-sum-less-eq-neg-numeral-iff
signed-real-of-word-sum-less-numeral-iff
signed-real-of-word-sum-less-neg-numeral-iff
numeral-less-eq-signed-real-of-word-sum
neg-numeral-less-eq-signed-real-of-word-sum
numeral-less-signed-real-of-word-sum
neg-numeral-less-signed-real-of-word-sum

declare *more-real-of-word-simps* [*simp*]

Interval-Word32.thy implements conservative interval arithmetic operators on 32-bit word values, with explicit infinities for values outside the representable bounds. It is suitable for use in interpreters for languages which must have a well-understood low-level behavior (see Interpreter.thy). This work was originally part of the paper by Bohrer *et al.* [2].

It is worth noting that this is not the first formalization of interval arithmetic in Isabelle/HOL. This article is presented regardless because it has unique goals in mind which have led to unique design decisions. Our goal is generate code which can be used to perform conservative arithmetic in implementations extracted from a proof.

The Isabelle standard library now features interval arithmetic, for example in Approximation.thy. Ours differs in two ways: 1) We use intervals with explicit positive and negative infinities, and with overflow checking. Such checking is often relevant in implementation-level code with unknown inputs. To promote memory-efficient implementations, we moreover use sentinel values for infinities, rather than datatype constructors. This is espe-

cially important in real-time settings where the garbarge collection required for datatypes can be a concern. 2) Our goal is not to use interval arithmetic to discharge Isabelle goals, but to generate useful proven-correct implementation code, see Interpreter.thy. On the other hand, we are not concerned with producing interval-based automation for arithmetic goals in HOL.

In practice, much of the work in this theory comes down to sheer case-analysis. Bounds-checking requires many edge cases in arithmetic functions, which come with many cases in proofs. Where possible, we attempt to offload interesting facts about word representations of numbers into reusable lemmas, but even then main results require many subcases, each with a certain amount of arithmetic grunt work.

1 Interval arithmetic definitions

1.1 Syntax

Words are 32-bit

type-synonym *word = 32 Word.word*

Sentinel values for infinities. Note that we leave the maximum value ($2^3 1$) completed unused, so that negation of $(2^{31}) - 1$ is not an edge case

definition *NEG-INF*::*word*
where *NEG-INF-def*[*simp*]:*NEG-INF* = −((*2* ^ *31*) −*1*)

definition *NegInf*::*real*
where *NegInf*[*simp*]:*NegInf* = *real-of-int* (*sint NEG-INF*)

definition *POS-INF*::*word*
where *POS-INF-def*[*simp*]:*POS-INF* = (*2*^*31*) − *1*

definition *PosInf*::*real*
where *PosInf*[*simp*]:*PosInf* = *real-of-int* (*sint POS-INF*)

Subtype of words who represent a finite value.

typedef *bword* = {*n*::*word*. *sint n* ≥ *sint NEG-INF* ∧ *sint n* ≤ *sint POS-INF*}
  apply(*rule exI*[**where** *x*=*NEG-INF*])
  by (*auto*)

Numeric literals

type-synonym *lit* = *bword*

setup-lifting *type-definition-bword*

lift-definition *bword-zero*::*bword* **is** *0*::*32 Word.word*
  by *auto*

lift-definition *bword-one*::*bword* **is** *1*::*32 Word.word*
by(*auto simp add*: *sint-uint*)

lift-definition *bword-neg-one*::*bword* **is** -1::*32 Word.word*
by(*auto*)

definition *word*::*word* ⇒ *bool*
where *word-def*[*simp*]:*word w* ≡ *w* ∈ {*NEG-INF*..*POS-INF*}

named-theorems *rep-simps Simplifications for representation functions*

Definitions of interval containment and word representation repe w r iff word w encodes real number r

inductive *repe* ::*word* ⇒ *real* ⇒ *bool* (**infix** ‹≡$_E$› *10*)
where
repPOS-INF:*r* ≥ *real-of-int* (*sint POS-INF*) ⟹ *repe POS-INF r*
| *repNEG-INF*:*r* ≤ *real-of-int* (*sint NEG-INF*) ⟹ *repe NEG-INF r*
| *repINT*:(*sint w*) < *real-of-int*(*sint POS-INF*) ⟹ (*sint w*) > *real-of-int*(*sint NEG-INF*)
⟹ *repe w* (*sint w*)

inductive-simps
repePos-simps[*rep-simps*]:*repe POS-INF r*
and *repeNeg-simps*[*rep-simps*]:*repe NEG-INF r*
and *repeInt-simps*[*rep-simps*]:*repe w* (*sint w*)

repU w r if w represents an upper bound of r

definition *repU* ::*word* ⇒ *real* ⇒ *bool* (**infix** ‹≡$_U$› *10*)
where *repU w r* ≡ ∃ *r'*. *r'* ≥ *r* ∧ *repe w r'*

lemma *repU-leq*:*repU w r* ⟹ *r'* ≤ *r* ⟹ *repU w r'*
unfolding *repU-def*
using *order-trans* **by** *auto*

repU w r if w represents a lower bound of r

definition *repL* ::*word* ⇒ *real* ⇒ *bool* (**infix** ‹≡$_L$› *10*)
where *repL w r* ≡ ∃ *r'*. *r'* ≤ *r* ∧ *repe w r'*

lemma *repL-geq*:*repL w r* ⟹ *r'* ≥ *r* ⟹ *repL w r'*
unfolding *repL-def*
using *order-trans* **by** *auto*

repP (l,u) r iff l and u encode lower and upper bounds of r

definition *repP* ::*word* ∗ *word* ⇒ *real* ⇒ *bool* (**infix** ‹≡$_P$› *10*)
where *repP w r* ≡ *let* (*w1*, *w2*) = *w in repL w1 r* ∧ *repU w2 r*

lemma *int-not-posinf*:
assumes *b1*:*real-of-int* (*sint ra*) < *real-of-int* (*sint POS-INF*)
assumes *b2*:*real-of-int* (*sint NEG-INF*) < *real-of-int* (*sint ra*)
shows *ra* ≠ *POS-INF*
using *b1 b2* **by** *auto*

lemma *int-not-neginf*:
assumes *b1*: *real-of-int* (*sint ra*) < *real-of-int* (*sint POS-INF*)
assumes *b2*: *real-of-int* (*sint NEG-INF*) < *real-of-int* (*sint ra*)
shows *ra* ≠ *NEG-INF*
using *b1 b2* **by** *auto*

lemma *int-not-undef*:
assumes *b1*:*real-of-int* (*sint ra*) < *real-of-int* (*sint POS-INF*)
assumes *b2*:*real-of-int* (*sint NEG-INF*) < *real-of-int* (*sint ra*)
shows *ra* ≠ *NEG-INF*−*1*
using *b1 b2* **by** *auto*

lemma *sint-range*:
assumes *b1*:*real-of-int* (*sint ra*) < *real-of-int* (*sint POS-INF*)
assumes *b2*:*real-of-int* (*sint NEG-INF*) < *real-of-int* (*sint ra*)
shows *sint ra* ∈ {*i*. *i* > −((*2*^*31*)−*1*) ∧ *i* < (*2*^*31*)−*1*}
using *b1 b2* **by** *auto*

lemma *word-size-neg*:
fixes *w* :: *32 Word.word*
shows *size* (−*w*) = *size w*
using *Word.word-size*[*of w*] *Word.word-size*[*of* −*w*] **by** *auto*

lemma *uint-distinct*:
fixes *w1 w2*
shows *w1* ≠ *w2* ⟹ *uint w1* ≠ *uint w2*
by *auto*

2 Preliminary lemmas

2.1 Case analysis lemmas

Case analysis principle for pairs of intervals, used in proofs of arithmetic operations

lemma *ivl-zero-case*:
fixes *l1 u1 l2 u2* :: *real*
assumes *ivl1*:*l1* ≤ *u1*
assumes *ivl2*:*l2* ≤ *u2*
shows
(*l1* ≤ *0* ∧ *0* ≤ *u1* ∧ *l2* ≤ *0* ∧ *0* ≤ *u2*)

$\vee(l1 \leq 0 \wedge 0 \leq u1 \wedge 0 \leq l2)$
$\vee(l1 \leq 0 \wedge 0 \leq u1 \wedge u2 \leq 0)$
$\vee(0 \leq l1 \wedge l2 \leq 0 \wedge 0 \leq u2)$
$\vee(u1 \leq 0 \wedge l2 \leq 0 \wedge 0 \leq u2)$
$\vee(u1 \leq 0 \wedge u2 \leq 0)$
$\vee(u1 \leq 0 \wedge 0 \leq l2)$
$\vee(0 \leq l1 \wedge u2 \leq 0)$
$\vee(0 \leq l1 \wedge 0 \leq l2)$
using *ivl1 ivl2*
by (*metis le-cases*)

lemma *case-ivl-zero*
[*consumes 2*, *case-names ZeroZero ZeroPos ZeroNeg PosZero NegZero NegNeg NegPos PosNeg PosPos*]:
fixes *l1 u1 l2 u2* :: *real*
shows
$l1 \leq u1 \Longrightarrow$
$l2 \leq u2 \Longrightarrow$
$((l1 \leq 0 \wedge 0 \leq u1 \wedge l2 \leq 0 \wedge 0 \leq u2) \Longrightarrow P) \Longrightarrow$
$((l1 \leq 0 \wedge 0 \leq u1 \wedge 0 \leq l2) \Longrightarrow P) \Longrightarrow$
$((l1 \leq 0 \wedge 0 \leq u1 \wedge u2 \leq 0) \Longrightarrow P) \Longrightarrow$
$((0 \leq l1 \wedge l2 \leq 0 \wedge 0 \leq u2) \Longrightarrow P) \Longrightarrow$
$((u1 \leq 0 \wedge l2 \leq 0 \wedge 0 \leq u2) \Longrightarrow P) \Longrightarrow$
$((u1 \leq 0 \wedge u2 \leq 0) \Longrightarrow P) \Longrightarrow$
$((u1 \leq 0 \wedge 0 \leq l2) \Longrightarrow P) \Longrightarrow$
$((0 \leq l1 \wedge u2 \leq 0) \Longrightarrow P) \Longrightarrow$
$((0 \leq l1 \wedge 0 \leq l2) \Longrightarrow P) \Longrightarrow P$
using *ivl-zero-case*[*of l1 u1 l2 u2*]
by *auto*

lemma *case-inf2*[*case-names PosPos PosNeg PosNum NegPos NegNeg NegNum NumPos NumNeg NumNum*]:
shows
$\bigwedge w1\ w2\ P.$
$(w1 = POS\text{-}INF \Longrightarrow w2 = POS\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow (w1 = POS\text{-}INF \Longrightarrow w2 = NEG\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow (w1 = POS\text{-}INF \Longrightarrow w2 \neq POS\text{-}INF \Longrightarrow w2 \neq NEG\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow (w1 = NEG\text{-}INF \Longrightarrow w2 = POS\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow (w1 = NEG\text{-}INF \Longrightarrow w2 = NEG\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow (w1 = NEG\text{-}INF \Longrightarrow w2 \neq POS\text{-}INF \Longrightarrow w2 \neq NEG\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow (w1 \neq POS\text{-}INF \Longrightarrow w1 \neq NEG\text{-}INF \Longrightarrow w2 = POS\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow (w1 \neq POS\text{-}INF \Longrightarrow w1 \neq NEG\text{-}INF \Longrightarrow w2 = NEG\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow (w1 \neq POS\text{-}INF \Longrightarrow w1 \neq NEG\text{-}INF \Longrightarrow w2 \neq POS\text{-}INF \Longrightarrow w2 \neq NEG\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow P\ w1\ w2$
by(*auto*)

lemma *case-pu-inf*[*case-names PosAny AnyPos NegNeg NegNum NumNeg NumNum*]:

shows
$\bigwedge w1\ w2\ P.$
$(w1 = POS\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow (w2 = POS\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow (w1 = NEG\text{-}INF \Longrightarrow w2 = NEG\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow (w1 = NEG\text{-}INF \Longrightarrow w2 \neq POS\text{-}INF \Longrightarrow w2 \neq NEG\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow (w1 \neq POS\text{-}INF \Longrightarrow w1 \neq NEG\text{-}INF \Longrightarrow w2 = NEG\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow (w1 \neq POS\text{-}INF \Longrightarrow w1 \neq NEG\text{-}INF \Longrightarrow w2 \neq POS\text{-}INF \Longrightarrow w2 \neq NEG\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow P\ w1\ w2$
by(*auto*)

lemma *case-pl-inf*[*case-names NegAny AnyNeg PosPos PosNum NumPos NumNum*]:
shows
$\bigwedge w1\ w2\ P.$
$(w1 = NEG\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow (w2 = NEG\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow (w1 = POS\text{-}INF \Longrightarrow w2 = POS\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow (w1 = POS\text{-}INF \Longrightarrow w2 \neq POS\text{-}INF \Longrightarrow w2 \neq NEG\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow (w1 \neq POS\text{-}INF \Longrightarrow w1 \neq NEG\text{-}INF \Longrightarrow w2 = POS\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow (w1 \neq POS\text{-}INF \Longrightarrow w1 \neq NEG\text{-}INF \Longrightarrow w2 \neq POS\text{-}INF \Longrightarrow w2 \neq NEG\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow P\ w1\ w2$
by(*auto*)

lemma *word-trichotomy*[*case-names Less Equal Greater*]:
fixes *w1 w2* :: *word*
shows
$(w1 <s\ w2 \Longrightarrow P\ w1\ w2) \Longrightarrow$
$(w1 = w2 \Longrightarrow P\ w1\ w2) \Longrightarrow$
$(w2 <s\ w1 \Longrightarrow P\ w1\ w2) \Longrightarrow P\ w1\ w2$
using *signed.linorder-cases* **by** *auto*

lemma *case-times-inf*
[*case-names*
PosPos NegPos PosNeg NegNeg
PosLo PosHi PosZero NegLo NegHi NegZero
LoPos HiPos ZeroPos LoNeg HiNeg ZeroNeg
AllFinite]:
fixes *w1 w2 P*
assumes *pp*:$(w1 = POS\text{-}INF \wedge w2 = POS\text{-}INF \Longrightarrow P\ w1\ w2)$
and *np*:$(w1 = NEG\text{-}INF \wedge w2 = POS\text{-}INF \Longrightarrow P\ w1\ w2)$
and *pn*:$(w1 = POS\text{-}INF \wedge w2 = NEG\text{-}INF \Longrightarrow P\ w1\ w2)$
and *nn*:$(w1 = NEG\text{-}INF \wedge w2 = NEG\text{-}INF \Longrightarrow P\ w1\ w2)$
and *pl*:$(w1 = POS\text{-}INF \wedge w2 \neq NEG\text{-}INF \wedge w2 <s\ 0 \Longrightarrow P\ w1\ w2)$
and *ph*:$(w1 = POS\text{-}INF \wedge w2 \neq POS\text{-}INF \wedge 0 <s\ w2 \Longrightarrow P\ w1\ w2)$
and *pz*:$(w1 = POS\text{-}INF \wedge w2 = 0 \Longrightarrow P\ w1\ w2)$
and *nl*:$(w1 = NEG\text{-}INF \wedge w2 \neq NEG\text{-}INF \wedge w2 <s\ 0 \Longrightarrow P\ w1\ w2)$

```
  and nh:(w1 = NEG-INF ∧ w2 ≠ POS-INF ∧ 0 <s w2 ⟹ P w1 w2)
  and nz:(w1 = NEG-INF ∧ 0 = w2 ⟹ P w1 w2)
  and lp:(w1 ≠ NEG-INF ∧ w1 <s 0 ∧ w2 = POS-INF ⟹ P w1 w2)
  and hp:(w1 ≠ POS-INF ∧ 0 <s w1 ∧ w2 = POS-INF ⟹ P w1 w2)
  and zp:(0 = w1 ∧ w2 = POS-INF ⟹ P w1 w2)
  and ln:(w1 ≠ NEG-INF ∧ w1 <s 0 ∧ w2 = NEG-INF ⟹ P w1 w2)
  and hn:(w1 ≠ POS-INF ∧ 0 <s w1 ∧ w2 = NEG-INF ⟹ P w1 w2)
  and zn:(0 = w1 ∧ w2 = NEG-INF ⟹ P w1 w2)
  and allFinite:w1 ≠ NEG-INF ∧ w1 ≠ POS-INF ∧ w2 ≠ NEG-INF ∧ w2 ≠
POS-INF ⟹ P w1 w2
  shows  P w1 w2
  proof (cases rule: word-trichotomy[of w1 0])
    case Less then have w1l:w1 <s 0 by auto
    then show ?thesis
    proof (cases rule: word-trichotomy[of w2 0])
      case Less
      then show ?thesis
        using nn nl ln nz zn allFinite w1l lp pl by blast
    next
      case Equal
      then show ?thesis
        using nn nl ln nz zn  w1l allFinite lp pz
        by (auto)
    next
      case Greater
      then show ?thesis
        using  nh np zp lp w1l allFinite ln ph
        by (auto)
    qed
  next
    case Equal then have w1z:w1 = 0 by auto
    then show ?thesis
    proof (cases rule: word-trichotomy[of w2 0])
      case Less
      then show ?thesis
        using zn allFinite w1z nl pl zp by auto
    next
      case Equal
      then show ?thesis
        using w1z allFinite
        by (auto)
    next
      case Greater
      then show ?thesis
        using allFinite zp w1z nh ph zn by auto
    qed
  next
    case Greater then have w1h:0 <s w1 by auto
    then show ?thesis
```

proof (*cases rule*: *word-trichotomy*[*of w2 0*])
case *Less*
then show *?thesis*
using *pn pl hn w1h allFinite hp nl* **by** *blast*
next
case *Equal*
then show *?thesis*
using *pz pz allFinite w1h hn hp nz* **by** *blast*
next
case *Greater*
then show *?thesis*
using *pp ph hp w1h allFinite hn nh* **by** *blast*
qed
qed

2.2 Trivial arithmetic lemmas

lemma *max-diff-pos*:$0 \leq 9223372034707292161 + ((-(2 \hat{} 31))::real)$ **by** *auto*

lemma *max-less*:$2 \hat{} 31 < (9223372039002259455::int)$ **by** *auto*

lemma *sints64*:$sints\ 64 = \{i. -(2 \hat{} 63) \leq i \wedge i < 2 \hat{} 63\}$
using *sints-def*[*of 64*] *range-sbintrunc*[*of 63*] **by** *auto*

lemma *sints32*:$sints\ 32 = \{i. -(2 \hat{} 31) \leq i \wedge i < 2 \hat{} 31\}$
using *sints-def*[*of 32*] *range-sbintrunc*[*of 31*] **by** *auto*

lemma *upcast-max*:$sint((scast(0x80000001::word))::64\ Word.word) = sint((0x80000001::32\ Word.word))$
by *auto*

lemma *upcast-min*:$(0xFFFFFFFF80000001::64\ Word.word) = ((scast\ (-0x7FFFFFFF::word))::64\ Word.word)$
by *auto*

lemma *min-extend-neg*:$sint\ ((0xFFFFFFFF80000001)::64\ Word.word) < 0$
by *auto*

lemma *min-extend-val'*:$sint\ ((-0x7FFFFFFF)::64\ Word.word) = (-0x7FFFFFFF)$
by *auto*

lemma *min-extend-val*:$(-0x7FFFFFFF::64\ Word.word) = 0xFFFFFFFF80000001$
by *auto*

lemma *range2s*:$\bigwedge x::int.\ x \leq 2 \hat{} 31 - 1 \Longrightarrow x + (-2147483647) < 2147483647$
by *auto*

3 Arithmetic operations

This section defines operations which conservatively compute upper and lower bounds of arithmetic functions given upper and lower bounds on their arguments. Each function comes with a proof that it rounds in the advertised direction.

3.1 Addition upper bound

Upper bound of w1 + w2

```
fun pu :: word ⇒ word ⇒ word
where pu w1 w2 =
(if w1 = POS-INF then POS-INF
 else if w2 = POS-INF then POS-INF
 else if w1 = NEG-INF then
  (if w2 = NEG-INF then NEG-INF
   else
    (let sum::64 Word.word = ((scast w2)::64 Word.word) + ((scast NEG-INF)::64
Word.word) in
    if (sum::64 Word.word) <=s ((scast NEG-INF)::64 Word.word) then NEG-INF
     else scast sum))
 else if w2 = NEG-INF then
  (let sum::64 Word.word = ((scast w1)::64 Word.word) + ((scast NEG-INF)::64
Word.word) in
   if (sum::64 Word.word) <=s ((scast NEG-INF)::64 Word.word) then NEG-INF
   else scast sum)
 else
  (let sum::64 Word.word = ((scast w1)::64 Word.word) + ((scast w2)::64 Word.word)
in
   if ((scast POS-INF)::64 Word.word) <=s (sum::64 Word.word) then POS-INF
   else if (sum::64 Word.word) <=s ((scast NEG-INF)::64 Word.word) then NEG-INF
    else scast sum))

lemma scast-down-range:
  fixes w::'a::len Word.word
  assumes sint w ∈ sints (len-of (TYPE('b::len)))
  shows sint w = sint ((scast w)::'b Word.word)
  using word-sint.Abs-inverse [OF assms] by simp

lemma pu-lemma:
  fixes w1 w2
  fixes r1 r2 :: real
  assumes up1:w1 ≡U (r1::real)
  assumes up2:w2 ≡U (r2::real)
  shows   pu w1 w2 ≡U (r1 + r2)
proof −
  have scast-eq1:sint((scast w1)::64 Word.word) = sint w1
    apply(rule Word.sint-up-scast)
```

```
  unfolding Word.is-up by auto
 have scast-eq2:sint((scast (0x80000001::word))::64 Word.word)
 = sint ((0x80000001::32 Word.word))
   by auto
 have scast-eq3:sint((scast w2)::64 Word.word) = sint w2
  apply(rule Word.sint-up-scast)
  unfolding Word.is-up by auto
 have w2Geq:sint ((scast w2)::64 Word.word) ≥ - (2 ^ 31)
  using word-sint.Rep[of (w2)::32 Word.word] sints32 Word.word-size
    scast-eq1 upcast-max scast-eq3 len32 mem-Collect-eq
  by auto
 have sint ((scast w2)::64 Word.word) ≤ 2 ^ 31
  apply (auto simp add: word-sint.Rep[of (w2)::32 Word.word] sints32
      scast-eq3 len32)
   using word-sint.Rep[of (w2)::32 Word.word] len32[of TYPE(32)] sints32 by
auto
 then have w2Less:sint ((scast w2)::64 Word.word) < 9223372039002259455 by
auto
 have w2Range:
  -(2 ^(size ((scast w2)::64 Word.word) - 1))
   ≤ sint ((scast w2)::64 Word.word) + sint ((-0x7FFFFFFF)::64 Word.word)
 ∧ sint ((scast w2)::64 Word.word) + sint ((-0x7FFFFFFF)::64 Word.word)
   ≤ 2 ^(size ((scast w2)::64 Word.word) - 1) - 1
  apply(auto simp add: Word.word-size scast-eq1 upcast-max sints64 max-less)
  using max-diff-pos max-less w2Less w2Geq by auto
 have w2case1a:sint (((scast w2)::64 Word.word) + (-0x7FFFFFFF::64 Word.word))

       = sint ((scast w2)::64 Word.word) + sint (-0x7FFFFFFF::64 Word.word)
  by(rule signed-arith-sint(1)[OF w2Range])
 have w1Lower:sint ((scast w1)::64 Word.word) ≥ - (2 ^ 31)
  using word-sint.Rep[of (w1)::32 Word.word] sints32 Word.word-size scast-eq1
scast-eq2
    scast-eq3 len32 mem-Collect-eq
  by auto
 have w1Leq:sint ((scast w1)::64 Word.word) ≤ 2 ^ 31
  apply (auto simp add: word-sint.Rep[of (w1)::32 Word.word] sints32 scast-eq1
len32)
   using word-sint.Rep[of (w1)::32 Word.word] len32[of TYPE(32)] sints32 by
auto
 then have w1Less:sint ((scast w1)::64 Word.word) < 9223372039002259455
  using max-less by auto
 have w1MinusBound: - (2 ^(size ((scast w1)::64 Word.word) - 1))
  ≤ sint ((scast w1)::64 Word.word) + sint ((-0x7FFFFFFF)::64 Word.word)
 ∧ sint ((scast w1)::64 Word.word) + sint ((-0x7FFFFFFF)::64 Word.word)
  ≤ 2 ^(size ((scast w1)::64 Word.word) - 1) - 1
  apply(auto simp add:
  Word.word-size[of (scast w1)::64 Word.word]
  Word.word-size[of (scast (-0x7FFFFFFF))::64 Word.word]
  scast-eq3 scast-eq2
```

```
    word-sint.Rep[of (w1)::32 Word.word]
    word-sint.Rep[of 0x80000001::32 Word.word]
    word-sint.Rep[of (scast w1)::64 Word.word]
    word-sint.Rep[of −0x7FFFFFFF::64 Word.word]
    sints64 sints32)
    using w1Lower w1Less by auto
 have w1case1a:sint (((scast w1)::64 Word.word) + (−0x7FFFFFFF::64 Word.word))

        = sint ((scast w1)::64 Word.word) + sint (−0x7FFFFFFF::64 Word.word)
    by (rule signed-arith-sint(1)[of (scast w1)::64 Word.word (− 0x7FFFFFFF),
OF w1MinusBound])
  have w1case1a′:sint (((scast w1)::64 Word.word) + 0xFFFFFFFF80000001)
      = sint ((scast w1)::64 Word.word) + sint ((−0x7FFFFFFF)::64 Word.word)
    using min-extend-val w1case1a by auto
  have w1Leq′:sint w1 ≤ 2^31 − 1
    using word-sint.Rep[of (w1)::32 Word.word]
    by (auto simp add: sints32 len32[of TYPE(32)])
  have neg64:(((scast w2)::64 Word.word) + 0xFFFFFFFF80000001)
         = ((scast w2)::64 Word.word) + (−0x7FFFFFFF) by auto
  have arith:⋀x::int. x ≤ 2 ^ 31 − 1 ⟹ x + (− 2147483647) < 2147483647
    by auto
  obtain r′1 and r′2 where
          geq1:r′1≥r1 and equiv1:w1 ≡E r′1
       and geq2:r′2≥r2 and equiv2:w2 ≡E r′2
        using up1 up2 unfolding repU-def by auto
  show ?thesis
proof (cases rule: case-pu-inf[where ?w1.0=w1, where ?w2.0=w2])
  case PosAny then show ?thesis
    apply (auto simp add: repU-def repe.simps)
    using linear by blast
next
  case AnyPos
  then show ?thesis
  apply (auto simp add: repU-def repe.simps)
  using linear by blast
next
  case NegNeg
  then show ?thesis
  using up1 up2
  by (auto simp add: repU-def repe.simps)
next
  case NegNum
  assume neq1:w2 ≠ POS-INF
  assume eq2:w1 = NEG-INF
  assume neq3:w2 ≠ NEG-INF
  let ?sum = (scast w2 + scast NEG-INF)::64 Word.word
  have leq1:r′1 ≤ (real-of-int (sint NEG-INF))
    using equiv1 neq1 eq2 neq3 by (auto simp add: repe.simps)
  have leq2:r′2 = (real-of-int (sint w2))
```

```
    using equiv2 neq1 eq2 neq3 by (auto simp add: repe.simps)
  have case1:?sum <=s ((scast NEG-INF)::64 Word.word) ⟹ NEG-INF ≡U r1
+ r2
    using up1 up2
    apply (simp add: repU-def repe.simps word-sle-eq)
    apply (rule exI [where x= r1 + r2])
    apply auto
    using w2case1a
    apply (auto simp add: eq2 scast-eq3)
    subgoal for r'
    proof −
      assume ‹r1 ≤ r'› ‹r' ≤ − 2147483647› ‹r2 ≤ signed w2› ‹sint w2 ≤ 0›
      from ‹sint w2 ≤ 0› have ‹real-of-int (sint w2) ≤ real-of-int 0›
        by (simp only: of-int-le-iff)
      then have ‹signed w2 ≤ (0::real)›
        by simp
      from ‹r1 ≤ r'› ‹r' ≤ − 2147483647› have ‹r1 ≤ − 2147483647›
        by (rule order-trans)
      moreover from ‹r2 ≤ signed w2› ‹signed w2 ≤ (0::real)› have ‹r2 ≤ 0›
        by (rule order-trans)
      ultimately show ‹r1 + r2 ≤ − 2147483647›
        by simp
    qed
    done
  have case2:¬(?sum <=s scast NEG-INF) ⟹ scast ?sum ≡U r1 + r2
    apply(simp add: repU-def repe.simps word-sle-def up1 up2)
    apply(rule exI[where x= r'2 − 0x7FFFFFFF])
    apply(rule conjI)
    subgoal
      proof −
        assume  ¬ sint (scast w2 + 0xFFFFFFFF80000001) ≤ − 2147483647
        have bound1:r1 ≤ − 2147483647
          using leq1 geq1 by (auto)
        have bound2:r2 ≤ r'2
          using leq2 geq2 by auto
        show r1 + r2 ≤ r'2 − 2147483647
          using bound1 bound2
          by(linarith)
      qed
    apply(rule disjI2)
    apply(rule disjI2)
    apply(auto simp add: not-le)
    subgoal
      proof −
        assume a:sint (((scast w2)::64 Word.word) + 0xFFFFFFFF80000001) >
− 2147483647
      then have sintw2-bound:sint (((scast w2)::64 Word.word) + (−0x7FFFFFFF))
> − 2147483647
          unfolding min-extend-val by auto
```

```
        have case1a: sint (((scast w2)::64 Word.word) + (−0x7FFFFFFF::64
Word.word))
                  =  sint ((scast w2)::64 Word.word) + sint (−0x7FFFFFFF::64
Word.word)
          by(rule signed-arith-sint(1)[OF w2Range])
        have − 0x7FFFFFFF < sint w2 + (− 0x7FFFFFFF)
          using sintw2-bound case1a min-extend-val′ scast-eq3 by linarith
        then have w2bound:0 < sint w2
          using less-add-same-cancel2 by blast
       have rightSize:sint (((scast w2)::64 Word.word) + − 0x7FFFFFFF) ∈ sints
(len-of TYPE(32))
       using case1a scast-eq3 min-extend-val′ word-sint.Rep[of (w2)::32 Word.word]
w2bound
          by (auto simp add: sints32 len32[of TYPE(32)])
      have downcast:sint ((scast (((scast w2)::64 Word.word) + ((− 0x7FFFFFFF))))::word)

                    = sint (((scast w2)::64 Word.word) + ((− 0x7FFFFFFF)::64
Word.word))
          using scast-down-range[OF rightSize]
          by auto
      then have b:sint ((scast (((scast w2)::64 Word.word) + 0xFFFFFFFF80000001))::word)
                = sint (((scast w2)::64 Word.word) + 0xFFFFFFFF80000001)
          using min-extend-val by auto
        have c:sint (((scast w2)::64 Word.word) + 0xFFFFFFFF80000001)
                 = sint ((scast w2)::64 Word.word) + sint ((−0x7FFFFFFF)::64
Word.word)
          using min-extend-val case1a by auto
        show ‹r′2 − 2147483647 = signed (SCAST(64 → 32) (SCAST(32 → 64)
w2 + 0xFFFFFFFF80000001))›
          using a b min-extend-val′ scast-eq3 leq2 case1a [symmetric]
          apply (simp add: algebra-simps)
          apply transfer
          apply simp
          done
      qed
    subgoal
    proof −
      have range2a: − (2 ^(size ((scast w2)::64 Word.word) − 1))
      ≤ sint ((scast w2)::64 Word.word) + sint ((−0x7FFFFFFF)::64 Word.word)

     ∧ sint ((scast w2)::64 Word.word) + sint ((−0x7FFFFFFF)::64 Word.word)
        ≤ 2 ^(size ((scast w2)::64 Word.word) − 1) − 1
        apply(auto simp add: Word.word-size scast-eq1 upcast-max sints64 sints32
max-less)
        using max-diff-pos max-less w2Geq w2Less by auto
     have case2a:sint (((scast w2)::64 Word.word) + (−0x7FFFFFFF::64 Word.word))

                  =  sint ((scast w2)::64 Word.word) + sint (−0x7FFFFFFF::64
Word.word)
```

```
      by(rule signed-arith-sint(1)[OF range2a])
    have neg64:(((scast w2)::64 Word.word) + 0xFFFFFFFF80000001)
          = ((scast w2)::64 Word.word) + (−0x7FFFFFFF) by auto
     assume sint (((scast w2)::64 Word.word) + 0xFFFFFFFF80000001) > −
2147483647
   then have sintw2-bound:sint (((scast w2)::64 Word.word) + (−0x7FFFFFFF))
> − 2147483647
      unfolding neg64 by auto
    have a:sint (((scast w2)::64 Word.word) + (−0x7FFFFFFF))
       = sint((scast w2)::64 Word.word) + sint((−0x7FFFFFFF)::64 Word.word)

      using case2a by auto
    have b:sint ((scast w2)::64 Word.word) = sint w2
      apply(rule Word.sint-up-scast)
      unfolding Word.is-up by auto
    have d:sint w2 ≤ 2^31 − 1
      using word-sint.Rep[of (w2)::32 Word.word]
      by (auto simp add: sints32 len32[of TYPE(32)])
    have − 0x7FFFFFFF < sint w2 + (− 0x7FFFFFFF)
      using sintw2-bound case2a min-extend-val′ scast-eq3 by linarith
    then have w2bound:0 < sint w2
      using less-add-same-cancel2 by blast
    have rightSize:sint (((scast w2)::64 Word.word) + − 0x7FFFFFFF) ∈ sints
(len-of TYPE(32))
      unfolding case2a b min-extend-val′
      using word-sint.Rep[of (w2)::32 Word.word] w2bound
      by (auto simp add: sints32 len32[of TYPE(32)])
   have downcast:sint ((scast (((scast w2)::64 Word.word) + ((− 0x7FFFFFFF))))::word)

                 = sint (((scast w2)::64 Word.word) + ((− 0x7FFFFFFF)::64
Word.word))
      using scast-down-range[OF rightSize]
      by auto
    have sint (scast (((scast w2)::64 Word.word) + (−0x7FFFFFFF))::word) <
2147483647
      unfolding downcast a b min-extend-val′
      using range2s[of sint w2, OF d]
      by auto
   then show sint (scast (((scast w2)::64 Word.word) + 0xFFFFFFFF80000001)::word)
< 2147483647
      by auto
   qed
   subgoal proof −
     assume notLeq:sint (((scast w2)::64 Word.word) + 0xFFFFFFFF80000001)
> − 2147483647
      then have gr:sint (((scast w2)::64 Word.word) + 0xFFFFFFFF80000001)
> − 2147483647
      by auto
   have case2a:sint (((scast w2)::64 Word.word) + (−0x7FFFFFFF::64 Word.word))
```

```
                = sint ((scast w2)::64 Word.word) + sint (−0x7FFFFFFF::64
Word.word)
      by(rule signed-arith-sint(1)[OF w2Range])
    from neg64
    have sintw2-bound:sint (((scast w2)::64 Word.word) + (−0x7FFFFFFF)) >
− 2147483647
      unfolding neg64 using notLeq by auto
    have a:sint (((scast w2)::64 Word.word) + (−0x7FFFFFFF))
      = sint((scast w2)::64 Word.word) + sint((−0x7FFFFFFF)::64 Word.word)

      using case2a by auto
    have c:sint((−0x7FFFFFFF)::64 Word.word) = −0x7FFFFFFF
      by auto
    have d:sint w2 ≤ 2^31 − 1
      using word-sint.Rep[of (w2)::32 Word.word]
      by (auto simp add: sints32 len32[of TYPE(32)])
    have − 0x7FFFFFFF < sint w2 + (− 0x7FFFFFFF)
      using sintw2-bound case2a c scast-eq3 by linarith
    then have w2bound:0 < sint w2
      using less-add-same-cancel2 by blast
    have rightSize:sint (((scast w2)::64 Word.word) + − 0x7FFFFFFF) ∈ sints
(len-of TYPE(32))
      unfolding case2a scast-eq3
      using word-sint.Rep[of (w2)::32 Word.word] w2bound
      by (auto simp add: sints32 len32[of TYPE(32)])
   have downcast:sint ((scast (((scast w2)::64 Word.word) + ((− 0x7FFFFFFF))))::word)

                = sint (((scast w2)::64 Word.word) + ((− 0x7FFFFFFF)::64
Word.word))
      using scast-down-range[OF rightSize]
      by auto
   have sintEq: sint ((scast (((scast w2)::64 Word.word) + 0xFFFFFFFF80000001))::word)

      = sint (((scast w2)::64 Word.word) + 0xFFFFFFFF80000001)
      using downcast by auto
    show − 2147483647 < sint (SCAST(64 → 32) (SCAST(32 → 64) w2 +
0xFFFFFFFF80000001))
      unfolding sintEq
      using gr of-int-less-iff of-int-minus of-int-numeral by linarith
    qed
  done
 have castEquiv:¬(?sum <=s scast NEG-INF) ⟹ (scast ?sum) ≡U r1 + r2
  using up1 up2 case1 case2 by fastforce
 have letRep:(let sum = ?sum in if sum <=s scast NEG-INF then NEG-INF else
scast sum) ≡U r1 + r2
  using case1 case2
  by(cases ?sum <=s scast NEG-INF; auto)
 show pu w1 w2 ≡U r1 + r2
```

```
    using letRep eq2 neq1
    by(auto)
next
  case NumNeg
  assume neq3:w1 ≠ NEG-INF
  assume neq1:w1 ≠ POS-INF
  assume eq2:w2 = NEG-INF
  let ?sum = (scast w1 + scast NEG-INF)::64 Word.word
  have case1:?sum <=s ((scast NEG-INF)::64 Word.word) ⟹ NEG-INF ≡U r1
+ r2
    using up1 up2 apply (simp add: repU-def repe.simps word-sle-def)
    apply(rule exI[where x= r1 + r2])
    apply(auto)
       using w1case1a min-extend-neg
     apply (auto simp add: neq1 eq2 neq3 repINT repU-def repe.simps repeInt-simps

       up2 word-sless-alt)
    using repINT repU-def repe.simps repeInt-simps up2 word-sless-alt
    proof −
     fix r'
     assume a1:sint ((scast w1)::64 Word.word) ≤ 0
     then have sint w1 ≤ 0 using scast-eq1 by auto
     then have h3: ‹signed w1 ≤ (0::real)›
       by transfer simp
     assume a2:r2 ≤ r'
     assume a3:r1 ≤ signed w1
     assume a4:r' ≤ (− 2147483647)
     from a2 a4 have h1:r2 ≤ − 2147483647 by auto
     from a1 a3 h3 have h2:r1 ≤ 0
     using dual-order.trans of-int-le-0-iff by blast
     show r1 + r2 ≤ (− 2147483647)
       using h1 h2 add.right-neutral add-mono by fastforce
    qed
  have leq1:r'2 ≤ (real-of-int (sint NEG-INF)) and leq2:r'1 = (real-of-int (sint
w1))
    using equiv1 equiv2 neq1 eq2 neq3 unfolding repe.simps by auto
 have case1a:sint (((scast w1)::64 Word.word) + (−0x7FFFFFFF::64 Word.word))

       = sint ((scast w1)::64 Word.word) + sint (−0x7FFFFFFF::64 Word.word)
    by(rule signed-arith-sint(1)[OF w1MinusBound])
  have case2:¬(?sum <=s scast NEG-INF) ⟹ scast ?sum ≡U r1 + r2
    apply (simp add: repU-def repe.simps word-sle-def up1 up2)
    apply(rule exI[where x= r'1 − 0x7FFFFFFF])
    apply(rule conjI)
    subgoal using leq1 leq2 geq1 geq2 by auto
    apply(rule disjI2)
    apply(rule disjI2)
    apply(auto)
    subgoal
```

```
      proof −
      have f:r'1 =  (real-of-int (sint w1))
        by (simp add: leq1 leq2 )
       assume a:¬ sint (((scast w1)::64 Word.word) + 0xFFFFFFFF80000001) ≤
− 2147483647
     then have sintw2-bound:sint (((scast w1)::64 Word.word) + (−0x7FFFFFFF))
> − 2147483647
        unfolding min-extend-val by auto
       have − 0x7FFFFFFF < sint w1 + (− 0x7FFFFFFF)
        using sintw2-bound case1a min-extend-val' scast-eq1 by linarith
       then have w2bound:0 < sint w1
        using less-add-same-cancel2 by blast
      have rightSize:sint (((scast w1)::64 Word.word) + − 0x7FFFFFFF) ∈ sints
(len-of TYPE(32))
        unfolding w1case1a
        using w2bound word-sint.Rep[of (w1)::32 Word.word]
        by (auto simp add: sints32 len32[of TYPE(32)] scast-eq1)
    have downcast:sint ((scast (((scast w1)::64 Word.word) + ((− 0x7FFFFFFF))))::word)
                    = sint (((scast w1)::64 Word.word) + ((− 0x7FFFFFFF)::64
Word.word))
        using scast-down-range[OF rightSize]
        by auto
    then have sint ((scast (((scast w1)::64 Word.word) + 0xFFFFFFFF80000001))::word)

                = sint (((scast w1)::64 Word.word) + 0xFFFFFFFF80000001)
        using min-extend-val by auto
    then have ‹signed (SCAST(64 → 32) (SCAST(32 → 64) w1 + 0xFFFFFFFF80000001))
=
        (signed (SCAST(32 → 64) w1 + 0xFFFFFFFF80000001) :: real)›
        by transfer simp
      moreover have r'1 − (real-of-int 2147483647) =
          (real-of-int (sint ((scast w1)::64 Word.word ) − 2147483647))
        by (simp add: scast-eq1 leq2)
      moreover from w1case1a'
      have ‹signed (SCAST(32 → 64) w1 + 0xFFFFFFFF80000001) =
       signed (SCAST(32 → 64) w1) + (signed (− 0x7FFFFFFF :: 64 Word.word)
:: real)›
        by transfer simp
      ultimately show r'1 −  2147483647
      = (signed ((scast (((scast w1)::64 Word.word) + 0xFFFFFFFF80000001))::word))
        by simp
    qed
    subgoal
    proof −
      assume ¬ sint (((scast w1)::64 Word.word) + 0xFFFFFFFF80000001) ≤ −
2147483647
    then have sintw2-bound:sint (((scast w1)::64 Word.word) + (−0x7FFFFFFF))
> − 2147483647
        unfolding neg64 by auto
```

have $-$ *0x7FFFFFFF* $<$ *sint w1* $+$ $(-$ *0x7FFFFFFF*$)$
using *sintw2-bound case1a min-extend-val′ scast-eq1* **by** *linarith*
then have *w2bound*:*0* $<$ *sint w1*
using *less-add-same-cancel2* **by** *blast*
have *rightSize*:*sint* $(((scast\ w1)::64\ Word.word) + -\ 0x7FFFFFFF) \in sints$ $(len\text{-}of\ TYPE(32))$
unfolding *case1a scast-eq1 w1case1a′*
using *word-sint.Rep*[*of* $(w1)::32\ Word.word$] *w2bound*
by(*auto simp add*: *sints32 len32*[*of TYPE*(*32*)])
have *downcast*:*sint* $((scast\ (((scast\ w1)::64\ Word.word) + ((-\ 0x7FFFFFFF))))::word)$
$= sint\ (((scast\ w1)::64\ Word.word) + ((-\ 0x7FFFFFFF)::64\ Word.word))$
using *scast-down-range*[*OF rightSize*]
by *auto*
show *sint* $(scast\ (((scast\ w1)::64\ Word.word) + 0xFFFFFFFF80000001)::word)$ < 2147483647
using *downcast min-extend-val′ w1case1a′ scast-eq1 arith*[*of sint w1*, *OF w1Leq′*]
by *auto*
qed
subgoal proof $-$
assume *notLeq*:$\neg$ *sint* $(((scast\ w1)::64\ Word.word) + 0xFFFFFFFF80000001)$ $\leq -\ 2147483647$
then have *gr*:*sint* $(((scast\ w1)::64\ Word.word) + 0xFFFFFFFF80000001)$ $> -\ 2147483647$
by *auto*
then have *sintw2-bound*:*sint* $(((scast\ w1)::64\ Word.word) + (-0x7FFFFFFF))$ $> -\ 2147483647$
unfolding *neg64* **using** *notLeq* **by** *auto*
have $-$ *0x7FFFFFFF* $<$ *sint w1* $+$ $(-$ *0x7FFFFFFF*$)$
using *sintw2-bound case1a min-extend-val′ scast-eq1* **by** *linarith*
then have *w2bound*:*0* $<$ *sint w1*
using *less-add-same-cancel2* **by** *blast*
have *rightSize*:*sint* $(((scast\ w1)::64\ Word.word) + -\ 0x7FFFFFFF) \in sints$ $(len\text{-}of\ TYPE(32))$
unfolding *case1a scast-eq1 w1case1a′*
using *word-sint.Rep*[*of* $(w1)::32\ Word.word$] *w2bound*
by (*auto simp add*: *sints32 len32*[*of TYPE*(*32*)])
show $-$ *2147483647*
$<$ *sint* $((scast\ (((scast\ w1)::64\ Word.word) + 0xFFFFFFFF80000001))::word)$
using *scast-down-range*[*OF rightSize*] *gr of-int-less-iff of-int-minus of-int-numeral*
by *simp*
qed
done
have *letUp*:$(let\ sum=?sum\ in\ if\ sum <=s\ scast\ NEG\text{-}INF\ then\ NEG\text{-}INF\ else\ scast\ sum) \equiv_U r1+r2$
using *case1 case2*
by (*auto simp add*: *Let-def*)

```
    have puSimp:pu w1 w2=(let sum = ?sum in if sum <=s scast NEG-INF then
NEG-INF else scast sum)
      using neq3 neq1 eq2 equiv1 leq2 repeInt-simps by force
    then show pu w1 w2 ≡U r1 + r2
      using letUp puSimp by auto
next
  case NumNum
  assume notinf1:w1 ≠ POS-INF
  assume notinf2:w2 ≠ POS-INF
  assume notneginf1:w1 ≠ NEG-INF
  assume notneginf2:w2 ≠ NEG-INF
  let ?sum = ((scast w1)::64 Word.word) + ((scast w2):: 64 Word.word)
  have inf-case:scast POS-INF <=s ?sum ⟹ POS-INF ≡U r1 + r2
    using repU-def repePos-simps
    by (meson dual-order.strict-trans not-less order-refl)
  have truth: − (2 ^(size ((scast w1)::64 Word.word) − 1))
     ≤ sint ((scast w1)::64 Word.word) + sint ((scast w2)::64 Word.word)
   ∧ sint ((scast w1)::64 Word.word) + sint ((scast w2)::64 Word.word)
     ≤ 2 ^(size ((scast w1)::64 Word.word) − 1) − 1
    using Word.word-size[of (scast w2)::64 Word.word]
      Word.word-size[of (scast w1)::64 Word.word]
    scast-eq1 scast-eq3
    word-sint.Rep[of (w1)::32 Word.word]
    word-sint.Rep[of (w2)::32 Word.word]
    word-sint.Rep[of (scast w1)::64 Word.word]
    word-sint.Rep[of (scast w2)::64 Word.word]
    sints64 sints32 by auto
  have sint-eq:sint((scast w1 + scast w2)::64 Word.word) = sint w1 + sint w2
   using signed-arith-sint(1)[of (scast w1)::64 Word.word (scast w2)::64 Word.word,
OF truth]
      scast-eq1 scast-eq3
    by auto
  have bigOne:scast w1 + scast w2 <=s ((− 0x7FFFFFFF)::64 Word.word)
    ⟹ ∃ r'≥r1 + r2. r' ≤ (− 0x7FFFFFFF)
   proof −
     assume scast w1 + scast w2 <=s ((− 0x7FFFFFFF)::64 Word.word)
     then have sint w1 + sint w2 ≤ − 0x7FFFFFFF
       using sint-eq unfolding word-sle-eq by auto
    then have sum-leq: ‹real-of-int (sint w1 + sint w2) ≤ real-of-int (− 0x7FFFFFFF)›
       by (simp only: of-int-le-iff)
     obtain r'1 r'2 ::real where
       bound1:r'1 ≥ r1 ∧ (w1 ≡E r'1) and
       bound2:r'2 ≥ r2 ∧ (w2 ≡E r'2)
       using up1 up2 unfolding repU-def by auto
     have somethingA:r'1 ≤ sint w1 and somethingB:r'2 ≤ sint w2
       using ‹scast w1 + scast w2 <=s − 0x7FFFFFFF› word-sle-def notinf1
notinf2
        bound1 bound2 unfolding repe.simps by auto
     have something:r'1 + r'2 ≤ sint w1 + sint w2
```

```
    using somethingA somethingB add-mono by fastforce
  show ∃r'≥r1 + r2. r' ≤ (- 0x7FFFFFFF)
    apply(rule exI[where x = r'1 + r'2])
    using bound1 bound2 add-mono something sum-leq
    apply (auto intro: order-trans [of - ‹signed-real-of-word w1 +
      signed-real-of-word w2›])
    done
 qed
 have anImp:⋀r'. (r'≥r1 + r2 ∧ r' ≤ (- 2147483647)) ⟹
  (∃r. - (2 ^ 31 - 1) = - (2 ^ 31 - 1)
   ∧ r' = r ∧ r ≤ (real-of-int (sint ((- (2 ^ 31 - 1))::32 Word.word))))
    by auto
 have anEq:((scast ((- (2 ^ 31 - 1))::32 Word.word))::64 Word.word) = (-
0x7FFFFFFF)
   by auto
 have bigTwo:
 ¬(((scast POS-INF)::64 Word.word) <=s ?sum) ⟹
  ¬(?sum <=s ((scast NEG-INF)::64 Word.word)) ⟹
  ∃r'≥r1 + r2. r' =
  (real-of-int (sint (scast (((scast w1)::64 Word.word) + ((scast w2)::64 Word.word))::word)))
  ∧ (r' < 0x7FFFFFFF ∧ (-0x7FFFFFFF) < r')
 proof -
   assume ¬(((scast POS-INF)::64 Word.word) <=s ?sum)
    and ¬(?sum <=s ((scast NEG-INF)::64 Word.word))
   then have interval-int: sint w1 + sint w2 < 0x7FFFFFFF
    (- 0x7FFFFFFF) < sint w1 + sint w2
    unfolding word-sle-eq POS-INF-def NEG-INF-def using sint-eq by auto
  then have interval: ‹real-of-int (sint w1 + sint w2) < real-of-int (0x7FFFFFFF)›
    ‹real-of-int (- 0x7FFFFFFF) < real-of-int (sint w1 + sint w2)›
    by (simp-all only: of-int-less-iff)
   obtain r'1 r'2 ::real where
    bound1:r'1 ≥ r1 ∧ (w1 ≡E r'1) and
    bound2:r'2 ≥ r2 ∧ (w2 ≡E r'2)
    using up1 up2 unfolding repU-def by auto
   have somethingA:r'1 ≤ sint w1 and somethingB:r'2 ≤ sint w2
    using word-sle-def notinf1 notinf2 bound1 bound2 unfolding repe.simps by
auto
   have something:r'1 + r'2 ≤ sint w1 + sint w2
    using somethingA somethingB add-mono by fastforce
   have (w1 ≡E r'1) using bound1 by auto
   then have
       r1w1:r'1 = (real-of-int (sint w1))
    and w1U: (real-of-int (sint w1)) < (real-of-int (sint POS-INF))
    and w1L: (real-of-int (sint NEG-INF)) < (real-of-int (sint w1))
    unfolding repe.simps
    using notinf1 notinf2 notneginf1 notneginf2 by (auto)
   have (w2 ≡E r'2) using bound2 by auto
   then have
       r2w1:r'2 = (real-of-int (sint w2))
```

```
    and w2U: (real-of-int (sint w2)) < (real-of-int (sint POS-INF))
    and w2L: (real-of-int (sint NEG-INF)) < (real-of-int (sint w2))
    unfolding repe.simps
    using notinf1 notinf2 notneginf1 notneginf2 by (auto)
  have sint (((scast w1)::64 Word.word) + ((scast w2)::64 Word.word))
   = sint ((scast (((scast w1)::64 Word.word) + ((scast w2)::64 Word.word)))::word)
    apply(rule scast-down-range)
    unfolding sint-eq using sints32 interval-int by auto
  then have cast-eq:(sint ((scast (((scast w1)::64 Word.word)
                + ((scast w2)::64 Word.word)))::word))
               = sint w1 + sint w2
    using scast-down-range sints32 interval-int sint-eq by auto
  from something and cast-eq
  have r12-sint-scast:r'1 + r'2
    = (sint ((scast (((scast w1)::64 Word.word)
      + ((scast w2)::64 Word.word)))::word))
    using r1w1 w1U w1L r2w1 w2U w2L by (simp)
  show ?thesis
    using bound1 bound2 add-mono r12-sint-scast cast-eq interval
    ‹r'1 + r'2 = (real-of-int (sint (scast (scast w1 + scast w2))))›
    by simp
qed
have neg-inf-case:?sum <=s ((scast ((NEG-INF)::word))::64 Word.word) ⟹
NEG-INF ≡U r1 + r2
proof (unfold repU-def NEG-INF-def repe.simps)
  assume scast w1 + scast w2 <=s ((scast ((- (2 ^ 31 - 1))::word))::64
Word.word)
  then have scast w1 + scast w2 <=s ((- 0x7FFFFFFF)::64 Word.word)
    by (metis anEq)
  then obtain r' where geq:(r' ≥ r1 + r2) and leq:(r' ≤ (- 0x7FFFFFFF))
   using bigOne by auto
  show (∃r'≥plus r1 r2.
    (∃r. uminus (minus(2 ^ 31) 1) = POS-INF ∧ r' = r ∧ (real-of-int (sint
POS-INF)) ≤ r)
  ∨ (∃r. uminus (minus(2 ^ 31) 1) = uminus (minus(2 ^ 31) 1)
    ∧ r' = r ∧ r ≤ real-of-int (sint ((uminus (minus(2 ^ 31) 1))::word)))
  ∨ (∃w. uminus (minus(2 ^ 31) 1) = w
    ∧ r' = real-of-int (sint w)
    ∧ (real-of-int (sint w)) < (real-of-int (sint POS-INF))
   ∧ less (real-of-int (sint (uminus (minus(2 ^ 31) 1)))) (real-of-int (sint w))))
    using leq anImp geq by meson
qed
have int-case:¬(((scast POS-INF)::64 Word.word) <=s ?sum)
          ⟹ ¬ (?sum <=s ((scast NEG-INF)::64 Word.word))
          ⟹ ((scast ?sum)::word) ≡U r1 + r2
proof −
  assume bound1:¬ ((scast POS-INF)::64 Word.word) <=s scast w1 + scast
w2
 assume bound2:¬ scast w1 + scast w2 <=s ((scast NEG-INF)::64 Word.word)
```

```
  obtain r'::real
    where rDef:r' = (real-of-int (sint ((scast (((scast w1)::64 Word.word)
                  + ((scast w2)::64 Word.word)))::word)))
    and r12:r'≥r1 + r2
    and boundU:r' <  0x7FFFFFFF
    and boundL:(−0x7FFFFFFF) < r'
    using bigTwo[OF bound1 bound2] by auto
  obtain w::word
 where wdef:w = (scast (((scast w1)::64 Word.word) + ((scast w2)::64 Word.word))::word)
    by auto
  then have wr:r' = (real-of-int (sint w))
    using r12 bound1 bound2 rDef by blast
  show ?thesis
    unfolding repU-def repe.simps
    using r12 wdef rDef boundU boundL wr
    by auto
qed
have almost:(let sum::64 Word.word = scast w1 + scast w2 in
  if scast POS-INF <=s sum then POS-INF
  else if sum <=s scast NEG-INF then NEG-INF
 else scast sum) ≡U r1 + r2
apply(cases ((scast POS-INF)::64 Word.word) <=s ((?sum)::64 Word.word))
subgoal using inf-case Let-def int-case neg-inf-case by auto
apply(cases ?sum <=s scast NEG-INF)
subgoal
  using inf-case Let-def int-case neg-inf-case
  proof −
    assume ¬ (scast POS-INF::64 Word.word) <=s scast w1 + scast w2
    then have ¬ (scast w1::64 Word.word) + scast w2 <=s scast NEG-INF
           ∧ ¬ (scast POS-INF::64 Word.word) <=s scast w1 + scast w2
           ∧ ¬ (scast w1::64 Word.word) + scast w2 <=s scast NEG-INF
        ∨ ((let w = scast w1 + scast w2 in
           if scast POS-INF <=s (w::64 Word.word)  then POS-INF
           else if w <=s scast NEG-INF then NEG-INF
           else scast w) ≡U r1 + r2)
      using neg-inf-case by presburger
    then show ?thesis
      using int-case by force
  qed
subgoal using inf-case Let-def int-case neg-inf-case
proof −
  assume a1: ¬ (scast POS-INF::64 Word.word) <=s scast w1 + scast w2
  assume ¬ (scast w1::64 Word.word) + scast w2 <=s scast NEG-INF
  have ¬ (scast w1::64 Word.word) + scast w2 <=s scast NEG-INF
     ∧ ¬ (scast POS-INF::64 Word.word) <=s scast w1 + scast w2
   ∨ ((let w = scast w1 + scast w2 in
     if scast POS-INF <=s (w::64 Word.word) then POS-INF
     else if w <=s scast NEG-INF then NEG-INF
     else scast w) ≡U r1 + r2)
```

```
      using a1 neg-inf-case by presburger
    then show ?thesis
      using int-case by force
  qed
  done
 then show ?thesis
   using notinf1 notinf2 notneginf1 notneginf2 by auto
qed
qed
```

Lower bound of w1 + w2

```
fun pl :: word ⇒ word ⇒ word
where pl w1 w2 =
(if w1 = NEG-INF then NEG-INF
 else if w2 = NEG-INF then NEG-INF
 else if w1 = POS-INF then
  (if w2 = POS-INF then POS-INF
   else
    (let sum::64 Word.word = ((scast w2)::64 Word.word) + ((scast POS-INF)::64
Word.word) in
    if ((scast POS-INF)::64 Word.word) <=s(sum::64 Word.word) then POS-INF
     else scast sum))
 else if w2 = POS-INF then
  (let sum::64 Word.word = ((scast w1)::64 Word.word) + ((scast POS-INF)::64
Word.word) in
   if ((scast POS-INF)::64 Word.word) <=s(sum::64 Word.word)  then POS-INF
   else scast sum)
 else
  (let sum::64 Word.word = ((scast w1)::64 Word.word) + ((scast w2)::64 Word.word)
in
   if ((scast POS-INF)::64 Word.word) <=s (sum::64 Word.word) then POS-INF
  else if (sum::64 Word.word) <=s ((scast NEG-INF)::64 Word.word) then NEG-INF
   else scast sum))
```

3.2 Addition lower bound

Correctness of lower bound of w1 + w2

```
lemma pl-lemma:
assumes lo1:w1 ≡L (r1::real)
assumes lo2:w2 ≡L (r2::real)
shows pl w1 w2 ≡L (r1 + r2)
proof −
  have scast-eq1:sint((scast w1)::64 Word.word) = sint w1
    apply(rule Word.sint-up-scast)
    unfolding Word.is-up by auto
 have scast-eq2:sint((scast (0x80000001::word))::64 Word.word)=sint((0x80000001::32
Word.word))
    by auto
  have scast-eq3:sint((scast w2)::64 Word.word) = sint w2
```

apply(*rule Word.sint-up-scast*)
unfolding *Word.is-up* **by** *auto*
have *sints64*:*sints 64* = {*i*. − (2 ^ 63) ≤ *i* ∧ *i* < 2 ^ 63}
using *sints-def*[*of 64*] *range-sbintrunc*[*of 63*] **by** *auto*
have *sints32*:*sints 32* = {*i*. − (2 ^ 31) ≤ *i* ∧ *i* < 2 ^ 31}
using *sints-def*[*of 32*] *range-sbintrunc*[*of 31*] **by** *auto*
have *thing1*:*0* ≤ *9223372034707292161* + ((−(2 ^ 31))::*real*) **by** *auto*
have *sint* ((*w2*)) ≥ (−(2 ^ 31))
using *word-sint.Rep*[*of* (*w2*)::*32 Word.word*] *sints32 mem-Collect-eq*
Word.word-size[*of* (*scast w2*)::*64 Word.word*] *scast-eq1 scast-eq2 scast-eq3 len32*

by *auto*
then have *thing4*:*sint* ((*scast w2*)::*64 Word.word*) ≥ (−(2 ^ 31))
using *scast-down-range sint-ge sints-num*
using *scast-eq3* **by** *linarith*
have *aLeq2*:(−(2 ^ 31)::*int*) ≥ −*9223372039002259455* **by** *auto*
then have *thing2*: (*0*::*int*) ≤ *9223372039002259455* + *sint* ((*scast w2*)::*64 Word.word*)
using *thing4 aLeq2*
by (*metis ab-group-add-class.ab-left-minus add.commute add-mono neg-le-iff-le*)
have *aLeq*:*2* ^ *31* ≤ (*9223372039002259455*::*int*) **by** *auto*
have *bLeq*:*sint* ((*scast w2*)::*64 Word.word*) ≤ *2* ^ *31*
apply (*auto simp add*: *word-sint.Rep*[*of* (*w2*)::*32 Word.word*] *sints32*
scast-eq3 len32)
using *word-sint.Rep*[*of* (*w2*)::*32 Word.word*] *len32*[*of TYPE*(*32*)] *sints32* **by** *auto*
have *thing3*: *sint* ((*scast w2*)::*64 Word.word*) ≤ *9223372034707292160*
using *aLeq bLeq* **by** *auto*
have *truth*: − (*2* ^(*size* ((*scast w2*)::*64 Word.word*) − *1*))
≤ *sint* ((*scast w2*)::*64 Word.word*) + *sint* ((*0x7FFFFFFF*)::*64 Word.word*)
∧ *sint* ((*scast w2*)::*64 Word.word*) + *sint* ((*0x7FFFFFFF*)::*64 Word.word*)
≤ *2* ^(*size* ((*scast w2*)::*64 Word.word*) − *1*) − *1*
by(*auto simp add*:
Word.word-size[*of* (*scast w2*)::*64 Word.word*]
Word.word-size[*of* (*scast* (*0x7FFFFFFF*))::*64 Word.word*]
scast-eq1 scast-eq2
sints64 sints32 thing2 thing1 thing3)
have *case1a*: *sint* (((*scast w2*)::*64 Word.word*) + (*0x7FFFFFFF*::*64 Word.word*))
= *sint* ((*scast w2*)::*64 Word.word*) + *sint* (*0x7FFFFFFF*::*64 Word.word*)
by(*rule signed-arith-sint*(*1*)[*OF truth*])
have *case1b*:*sint* ((*0xFFFFFFFF80000001*)::*64 Word.word*) < *0*
by *auto*
have *arith*:⋀*x*::*int*. *x* ≤ *2* ^ *31* − *1* ⟹ *x* + (− *2147483647*) < *2147483647*
by *auto*
have *neg64*:(((*scast w2*)::*64 Word.word*) + *0x7FFFFFFF*)
= ((*scast w2*)::*64 Word.word*) + (*0x7FFFFFFF*)
by *auto*
obtain r'_1 **and** r'_2 **where**

```
        geq1:r'1≤r1 and equiv1:w1 ≡E r'1
      and geq2:r'2≤r2 and equiv2:w2 ≡E r'2
       using lo1 lo2 unfolding repL-def by auto
 show ?thesis
proof (cases rule: case-pl-inf[where ?w1.0=w1, where ?w2.0=w2])
 case NegAny
 then show ?thesis
 apply (auto simp add: repL-def repe.simps)
 using lo1 lo2 linear by auto
next
 case AnyNeg
 then show ?thesis
 apply (auto simp add: repL-def repe.simps)
 using linear by auto
next
 case PosPos
 then show ?thesis
 using lo1 lo2
 by (auto simp add: repL-def repe.simps)
next
 case PosNum
 assume neq1:w2 ≠ POS-INF
 assume eq2:w1 = POS-INF
 assume neq3:w2 ≠ NEG-INF
 let ?sum = (scast w2 + scast POS-INF)::64 Word.word
 have case1:(((scast POS-INF)::64 Word.word) <=s ?sum) ⟹ POS-INF ≡L r1
+ r2
   using lo1 lo2 apply (simp add: repL-def repe.simps word-sle-def)
   apply(rule exI[where x= r1 + r2])
   using case1a case1b
   apply (auto simp add: neq1 eq2 neq3 repINT repL-def repe.simps
     repeInt-simps lo2 word-sless-alt)
  proof −
    fix r'
    assume a1:0 ≤ sint (((scast w2)::64 Word.word))
    from a1 have h3:2147483647 ≤ sint w2 + 0x7FFFFFFF using scast-eq3
      by auto
    assume a2:r' ≤ r1
    assume a3:signed w2 ≤ r2
    assume a4:(2147483647) ≤ r'
    from a2 a4 have h1:2147483647 ≤ r1 by auto
    from a1 a3 h3 have h2:0 ≤ r2
      using of-int-le-0-iff le-add-same-cancel2
      apply simp
      apply transfer
      apply simp
    apply (metis (full-types) of-int-0 of-int-le-iff order-trans signed-take-bit-nonnegative-iff)
      done
    show (2147483647) ≤ r1 + r2
```

```
      using h1 h2 h3 add.right-neutral add-mono
      by fastforce
    qed
  have leq1:r'1 ≥ (real-of-int (sint POS-INF))
    using equiv1 neq1 eq2 neq3
    unfolding repe.simps by auto
  have leq2:r'2 = (real-of-int (sint w2))
    using equiv2 neq1 eq2 neq3
    unfolding repe.simps by auto
  have case2:¬(scast POS-INF <=s ?sum) ⟹ scast ?sum ≡L r1 + r2
    apply (simp add: repL-def repe.simps word-sle-def lo1 lo2)
    apply(rule exI[where x= r'2 + 0x7FFFFFFF])
    apply(rule conjI)
    subgoal
     proof −
      assume ¬ 2147483647 ≤ sint (scast w2 + 0x7FFFFFFF)
      have bound1:2147483647 ≤ r1
        using leq1 geq1 by (auto)
      have bound2:r'2 ≤ r2
        using leq2 geq2 by auto
      show r'2 + 2147483647 ≤ r1 + r2
        using bound1 bound2
        by linarith
     qed
    apply(rule disjI2)
    apply(rule disjI2)
    apply(auto)
    subgoal
     proof −
    assume a:¬ 2147483647 ≤ sint (((scast w2)::64 Word.word) + 0x7FFFFFFF)
     then have sintw2-bound:2147483647 > sint (((scast w2)::64 Word.word) +
(0x7FFFFFFF))
       by auto
    have case1a:sint (((scast w2)::64 Word.word) + (0x7FFFFFFF::64 Word.word))

                  =  sint ((scast w2)::64 Word.word) + sint (0x7FFFFFFF::64
Word.word)
       by(rule signed-arith-sint(1)[OF truth])
      have a1:sint (((scast w2)::64 Word.word) + (0x7FFFFFFF))
         = sint((scast w2)::64 Word.word) + sint((0x7FFFFFFF)::64 Word.word)

       using case1a by auto
      have c1:sint((0x7FFFFFFF)::64 Word.word) = 0x7FFFFFFF
       by auto
      have sint w2 + ( 0x7FFFFFFF) <  0x7FFFFFFF
       using sintw2-bound case1a c1 scast-eq3 by linarith
      then have w2bound:sint w2 < 0
       using add-less-same-cancel2 by blast
      have rightSize:sint (((scast w2)::64 Word.word) +  0x7FFFFFFF) ∈ sints
```

```
(len-of TYPE(32))
        unfolding case1a scast-eq3 c1
        using word-sint.Rep[of (w2)::32 Word.word] w2bound
        by (auto simp add: sints32 len32[of TYPE(32)])
    have downcast:sint ((scast (((scast w2)::64 Word.word) + (( 0x7FFFFFFF))))::word)
       = sint (((scast w2)::64 Word.word) + (( 0x7FFFFFFF)::64 Word.word))
       using scast-down-range[OF rightSize]
       by auto
    then have b:sint ((scast (((scast w2)::64 Word.word) + 0x7FFFFFFF))::word)
              = sint (((scast w2)::64 Word.word) + 0x7FFFFFFF)
       by auto
     have c:sint (((scast w2)::64 Word.word) + 0x7FFFFFFF)
       = sint ((scast w2)::64 Word.word) + sint ((0x7FFFFFFF)::64 Word.word)
       using case1a by auto
     have d:sint ((0x7FFFFFFF)::64 Word.word) = (0x7FFFFFFF) by auto
     have f:r'2 =  (real-of-int (sint w2))
       by (simp add: leq2)
     show r'2 +  2147483647
       = (signed ((scast (((scast w2)::64 Word.word) + 0x7FFFFFFF))::word))
       using a b c d scast-eq3 f leq2 of-int-numeral
       by auto
     qed
   subgoal
   proof −
     have truth2a:−(2^(size ((scast w2)::64 Word.word)−1))
       ≤ sint ((scast w2)::64 Word.word) + sint ((0x7FFFFFFF)::64 Word.word)

      ∧ sint ((scast w2)::64 Word.word) + sint ((0x7FFFFFFF)::64 Word.word)
        ≤ 2 ^ (size ((scast w2)::64 Word.word) − 1) − 1
      apply(auto simp add:
          Word.word-size[of (scast w2)::64 Word.word]
          Word.word-size[of (scast (0x7FFFFFFF))::64 Word.word]
          scast-eq1 scast-eq2 sints64 sints32 thing2)
      using thing1 thing2 thing3 by auto
   have case2a: sint (((scast w2)::64 Word.word) + (0x7FFFFFFF::64 Word.word))

                    = sint ((scast w2)::64 Word.word) + sint (0x7FFFFFFF::64
Word.word)
       by(rule signed-arith-sint(1)[OF truth2a])
   have min-cast:(0x7FFFFFFF::64 Word.word) =((scast (0x7FFFFFFF::word))::64
Word.word)
       by auto
    assume ¬ 2147483647 ≤ sint (((scast w2)::64 Word.word) + 0x7FFFFFFF)
     then have sintw2-bound:2147483647 > sint (((scast w2)::64 Word.word) +
(0x7FFFFFFF))
       using neg64 by auto
     have a:sint (((scast w2)::64 Word.word) + (0x7FFFFFFF))
        = sint((scast w2)::64 Word.word) + sint((0x7FFFFFFF)::64 Word.word)
```

using *case2a* **by** *auto*
have *c*:*sint*((*0x7FFFFFFF*)::*64 Word.word*) = *0x7FFFFFFF*
by *auto*
have *0x7FFFFFFF* > *sint w2* + (*0x7FFFFFFF*)
using *sintw2-bound case2a c scast-eq3* **by** *linarith*
then have *w2bound*: *sint w2* < *0*
by *simp*
have *rightSize*:*sint* (((*scast w2*)::*64 Word.word*) + *0x7FFFFFFF*) ∈ *sints* (*len-of TYPE*(*32*))
unfolding *case2a scast-eq3 c*
apply (*auto simp add*: *sints32 len32*[*of TYPE*(*32*)])
using *word-sint.Rep*[*of* (*w2*)::*32 Word.word*] *sints32 len32*[*of TYPE*(*32*)] *w2bound*
by *auto*
have *downcast*:*sint* ((*scast* (((*scast w2*)::*64 Word.word*) + ((*0x7FFFFFFF*))))::*word*)
= *sint* (((*scast w2*)::*64 Word.word*) + ((*0x7FFFFFFF*)::*64 Word.word*))
using *scast-down-range*[*OF rightSize*]
by *auto*
then show *sint* (*scast* (((*scast w2*)::*64 Word.word*) + *0x7FFFFFFF*)::*word*) < *2147483647*
unfolding *downcast a scast-eq3 c*
using *w2bound* **by** *auto*
qed
subgoal proof −
assume *notLeq*:¬ *2147483647* ≤ *sint* (((*scast w2*)::*64 Word.word*) + *0x7FFFFFFF*)
then have *gr*:*sint* (((*scast w2*)::*64 Word.word*) + *0x7FFFFFFF*) < *2147483647*
by *auto*
have *case2a*: *sint* (((*scast w2*)::*64 Word.word*) + (*0x7FFFFFFF*::*64 Word.word*))
= *sint* ((*scast w2*)::*64 Word.word*) + *sint* (*0x7FFFFFFF*::*64 Word.word*)
by(*rule signed-arith-sint*(*1*)[*OF truth*])
have *min-cast*:(*0x7FFFFFFF*::*64 Word.word*) =((*scast* (*0x7FFFFFFF*::*word*))::*64 Word.word*)
by *auto*
have *neg64*:(((*scast w2*)::*64 Word.word*) + *0x7FFFFFFF*)
= ((*scast w2*)::*64 Word.word*) + (*0x7FFFFFFF*)
by *auto*
then have *sintw2-bound*:*sint* (((*scast w2*)::*64 Word.word*) + (*0x7FFFFFFF*)) < *2147483647*
using *neg64* **using** *notLeq* **by** *auto*
have *a*:*sint* (((*scast w2*)::*64 Word.word*) + (*0x7FFFFFFF*))
= *sint*((*scast w2*)::*64 Word.word*) + *sint*((*0x7FFFFFFF*)::*64 Word.word*)
using *case2a* **by** *auto*
have *c*:*sint*((*0x7FFFFFFF*)::*64 Word.word*) = *0x7FFFFFFF*
by *auto*
have − *2147483647* ≠ *w2* **using** *neq3* **unfolding** *NEG-INF-def* **by** *auto*

```
      then have sint((− 2147483647)::word) ≠ sint w2
        using word-sint.Rep-inject by blast
      then have n1:− 2147483647 ≠ sint w2
        by auto
      have − 2147483648 ≠ w2
        apply(rule repe.cases[OF equiv2])
        by auto
      then have sint(− 2147483648::word) ≠ sint w2
      using word-sint.Rep-inject by blast
      then have n2:− 2147483648 ≠ sint w2
        by auto
      then have d:sint w2 > − 2147483647
         using word-sint.Rep[of (w2)::32 Word.word] sints32 len32[of TYPE(32)]
neq3 n1 n2
        by auto
      have w2bound:− 2147483647 < sint w2 + 0x7FFFFFFF
        using sintw2-bound case2a c scast-eq3 d by linarith
       have rightSize:sint (((scast w2)::64 Word.word) +  0x7FFFFFFF) ∈ sints
(len-of TYPE(32))
          using sints32 len32[of TYPE(32)] w2bound word-sint.Rep[of (w2)::32
Word.word]
          c case2a scast-eq3 sintw2-bound
        by (auto simp add: sints32 len32[of TYPE(32)])
    have downcast:sint ((scast (((scast w2)::64 Word.word) + (( 0x7FFFFFFF))))::word)
                    = sint (((scast w2)::64 Word.word) + (( 0x7FFFFFFF)::64
Word.word))
        using scast-down-range[OF rightSize]
        by auto
    have sintEq: sint ((scast (((scast w2)::64 Word.word) + 0x7FFFFFFF))::word)
=
          sint (((scast w2)::64 Word.word) + 0x7FFFFFFF)
          using downcast by auto
      show − 2147483647
          < sint ((scast (((scast w2)::64 Word.word) + 0x7FFFFFFF))::word)
        unfolding sintEq
       using gr of-int-less-iff of-int-minus of-int-numeral c case2a scast-eq3 w2bound
        by simp
      qed
      done
  have (let sum = ?sum in if scast POS-INF <=s sum then POS-INF else scast
sum) ≡L r1 + r2
    using case1 case2
    by (auto simp add: Let-def)
  then show ?thesis
    using lo1 lo2 neq1 eq2 neq3
    by (auto)
next
  case NumPos
    assume neq3:w1 ≠ NEG-INF
```

assume *neq1*:*w1* ≠ *POS-INF*
assume *eq2*:*w2* = *POS-INF*
let *?sum* = (*scast w1* + *scast POS-INF*)::*64 Word.word*
have *thing1*:*0* ≤ *9223372034707292161* + ((−(*2* ^ *31*))::*real*) **by** *auto*
have *sint* ((*w1*)) ≥ (−(*2* ^ *31*))
using *word-sint.Rep*[*of* (*w1*)::*32 Word.word*] *scast-eq1 scast-eq2 scast-eq3*
Word.word-size[*of* (*scast w1*)::*64 Word.word*] *sints32 len32 mem-Collect-eq*
by *auto*
then have *thing4*:*sint* ((*scast w1*)::*64 Word.word*) ≥ (−(*2* ^ *31*))
using *scast-down-range sint-ge sints-num scast-eq3 scast-eq1* **by** *linarith*
have *aLeq2*:(−(*2* ^ *31*)::*int*) ≥ −*922337203902259455* **by** *auto*
then have *thing2*: (*0*::*int*) ≤ *922337203902259455* + *sint* ((*scast w1*)::*64 Word.word*)
using *thing4 aLeq2*
by (*metis ab-group-add-class.ab-left-minus add.commute add-mono neg-le-iff-le*)
have *aLeq*:*2* ^ *31* ≤ (*922337203902259455*::*int*) **by** *auto*
have *bLeq*:*sint* ((*scast w1*)::*64 Word.word*) ≤ *2* ^ *31*
apply (*auto simp add*: *word-sint.Rep*[*of* (*w1*)::*32 Word.word*] *sints32*
scast-eq1 len32)
using *word-sint.Rep*[*of* (*w1*)::*32 Word.word*] *len32*[*of TYPE*(*32*)] *sints32*
by *clarsimp*
have *thing3*: *sint* ((*scast w1*)::*64 Word.word*) ≤ *9223372034707292160*
using *aLeq bLeq* **by** *auto*
have *truth*: − (*2* ^ (*size* ((*scast w1*)::*64 Word.word*) − *1*))
≤ *sint* ((*scast w1*)::*64 Word.word*) + *sint* ((*0x7FFFFFFF*)::*64 Word.word*)
∧ *sint* ((*scast w1*)::*64 Word.word*) + *sint* ((*0x7FFFFFFF*)::*64 Word.word*)
≤ *2* ^ (*size* ((*scast w1*)::*64 Word.word*) − *1*) − *1*
by(*auto simp add*:
Word.word-size[*of* (*scast w1*)::*64 Word.word*]
Word.word-size[*of* (*scast* (*0x7FFFFFFF*))::*64 Word.word*]
scast-eq3 scast-eq2 sints64 sints32 thing2 thing1 thing3)
have *case1a*:*sint* (((*scast w1*)::*64 Word.word*) + (*0x7FFFFFFF*::*64 Word.word*))
= *sint* ((*scast w1*)::*64 Word.word*) + *sint* (*0x7FFFFFFF*::*64 Word.word*)
by(*rule signed-arith-sint*(*1*)[*OF truth*])
have *case1b*:*sint* ((*0xFFFFFFFF80000001*)::*64 Word.word*) < *0*
by *auto*
have *g*:(*0x7FFFFFFF*::*64 Word.word*) = *0x7FFFFFFF* **by** *auto*
have *c*:*sint* (((*scast w1*)::*64 Word.word*) + *0x7FFFFFFF*)
= *sint* ((*scast w1*)::*64 Word.word*) + *sint* ((*0x7FFFFFFF*)::*64 Word.word*)
using *g case1a*
by *blast*
have *d*:*sint* ((*0x7FFFFFFF*)::*64 Word.word*) = (*0x7FFFFFFF*) **by** *auto*
have *e*:*sint* ((*scast w1*)::*64 Word.word*) = *sint w1*
using *scast-eq1* **by** *blast*
have *d2*:*sint w1* ≤ *2*^*31* − *1*
using *word-sint.Rep*[*of* (*w1*)::*32 Word.word*]

```
  by (auto simp add: sints32 len32[of TYPE(32)])
 have case1:scast POS-INF <=s ?sum ⟹ POS-INF ≡L r1 + r2
  using lo1 lo2 apply (simp add: repL-def repe.simps word-sle-def)
  apply(rule exI[where x= r1 + r2])
  apply(auto)
    using case1a case1b
    apply (auto simp add: neq1 eq2 neq3 repINT repL-def
      repe.simps repeInt-simps lo2 word-sless-alt)
 proof −
   fix r'
   have h3:sint (((scast w1)::64 Word.word) + 0x7FFFFFFF)
    = sint (((scast w1)::64 Word.word)) + sint(0x7FFFFFFF::64 Word.word)

     using case1a by auto
   have h4:sint(0x7FFFFFFF::64 Word.word) = 2147483647 by auto
   assume a1:0 ≤ sint ((scast w1)::64 Word.word)
   then have h3:sint w1 ≥ 0 using scast-eq1 h3 h4 by auto
   assume a2:r' ≤ r2
   assume a3:signed w1 ≤ r1
   assume a4:(2147483647) ≤ r'
   from a2 a4 have h1:r2 ≥ 2147483647 by auto
   from a3 h3 have h2:r1 ≥ 0
     by (auto intro: order-trans [of - ‹signed-real-of-word w1›])
   show 2147483647 ≤ r1 + r2
     using h1 h2 add.right-neutral add-mono by fastforce
   qed
 have leq1:r'2 ≥ (real-of-int (sint POS-INF)) and leq2:r'1 = (real-of-int (sint
w1))
   using equiv1 equiv2 neq1 eq2 neq3 unfolding repe.simps by auto
 have neg64:(((scast w1)::64 Word.word) + 0xFFFFFFFF80000001)
        = ((scast w1)::64 Word.word) + (−0x7FFFFFFF)
  by auto
 have case2:¬(scast POS-INF <=s ?sum) ⟹ scast ?sum ≡L r1 + r2
  apply (simp add: repL-def repe.simps word-sle-def lo1 lo2)
  apply(rule exI[where x= r'1 + 0x7FFFFFFF])
  apply(rule conjI)
  subgoal
    proof −
     assume ¬ 2147483647 ≤ sint (scast w1 + 0x7FFFFFFF)
     have bound1:r2 ≥ 2147483647
       using leq1 geq2 by (auto)
     have bound2:r1 ≥ r'1
       using leq2 geq1 by auto
     show r'1 + 2147483647 ≤ r1 + r2
       using bound1 bound2
       by linarith
     qed
  apply(rule disjI2)
  apply(rule disjI2)
```

```
      apply(auto)
      subgoal
        proof −
        have f:r'1 =  (real-of-int (sint w1))
          by (simp add: leq1 leq2 )
       assume a:¬ 2147483647 ≤ sint (((scast w1)::64 Word.word) + 0x7FFFFFFF)
          then have sintw2-bound:2147483647 > sint (((scast w1)::64 Word.word)
+ (0x7FFFFFFF))
           by auto
         have 0x7FFFFFFF > sint w1 + (0x7FFFFFFF)
           using sintw2-bound case1a d scast-eq1 by linarith
         then have w2bound:0 > sint w1
           using add-less-same-cancel2 by blast
         have rightSize:sint (((scast w1)::64 Word.word) + 0x7FFFFFFF) ∈ sints
(len-of TYPE(32))
           unfolding case1a e
           using w2bound word-sint.Rep[of (w1)::32 Word.word]
           by (auto simp add: sints32 len32[of TYPE(32)] )
       have arith:⋀x::int. x ≤ 2 ^ 31 − 1 ⟹ x + (− 2147483647) < 2147483647
          by auto
      have downcast:sint ((scast (((scast w1)::64 Word.word) + (( 0x7FFFFFFF))))::word)

                          = sint (((scast w1)::64 Word.word) + ((  0x7FFFFFFF)::64
Word.word))
           using scast-down-range[OF rightSize]
           by auto
      then have b:sint((scast (((scast w1)::64 Word.word) + 0x7FFFFFFF))::word)
                  = sint(((scast w1)::64 Word.word) + 0x7FFFFFFF)
          using g by auto
        show r'1 + 2147483647
       = ((signed-real-of-word ((scast (((scast w1)::64 Word.word) + 0x7FFFFFFF))::word)))
          using a b c d e f
          proof −
            have (real-of-int (sint ((scast w1)::64 Word.word ) + 2147483647))
                = r'1 +  (real-of-int 2147483647)
              using e leq2 by auto
            from this [symmetric] show ?thesis
              using b c d of-int-numeral
              by simp
          qed
        qed
      subgoal
      proof −
       assume ¬ 2147483647 ≤ sint (((scast w1)::64 Word.word) + 0x7FFFFFFF)
      then have sintw2-bound:sint (((scast w1)::64 Word.word) + (0x7FFFFFFF))
< 2147483647
          unfolding neg64 by auto
        have 0x7FFFFFFF > sint w1 + (0x7FFFFFFF)
          using sintw2-bound case1a d scast-eq1 by linarith
```

```
      then have w2bound:0 > sint w1
        using add-less-same-cancel2 by blast
       have rightSize:sint (((scast w1)::64 Word.word) + 0x7FFFFFFF) ∈ sints
(len-of TYPE(32))
        unfolding case1a e c
        using word-sint.Rep[of (w1)::32 Word.word] w2bound
        by (auto simp add: sints32 len32[of TYPE(32)])
      have arith:⋀x::int. x ≤ 2 ^ 31 − 1 ⟹ x + (− 2147483647) < 2147483647
        by auto
     have downcast:sint ((scast (((scast w1)::64 Word.word) + 0x7FFFFFFF))::word)

        = sint (((scast w1)::64 Word.word) + ((0x7FFFFFFF)::64 Word.word))
        using scast-down-range[OF rightSize]
        by auto
       show sint (scast (((scast w1)::64 Word.word) + 0x7FFFFFFF)::word) <
2147483647
        using downcast d e c arith[of sint w1, OF d2] sintw2-bound by linarith
     qed
     subgoal proof −
         assume notLeq:¬ 2147483647 ≤ sint (((scast w1)::64 Word.word) +
0x7FFFFFFF)
     then have gr:2147483647 > sint (((scast w1)::64 Word.word) + 0x7FFFFFFF)

        by auto
     then have sintw2-bound:sint (((scast w1)::64 Word.word) + (0x7FFFFFFF))
< 2147483647
        unfolding neg64 using notLeq by auto
       have 0x7FFFFFFF > sint w1 + ( 0x7FFFFFFF)
        using sintw2-bound case1a d scast-eq1 by linarith
       then have useful:0 > sint w1
        using add-less-same-cancel2 by blast
       have rightSize:sint (((scast w1)::64 Word.word) +  0x7FFFFFFF) ∈ sints
(len-of TYPE(32))
        unfolding case1a e
        using word-sint.Rep[of (w1)::32 Word.word]
          sints32 len32[of TYPE(32)] useful
        by auto
       have − 2147483647 ≠ w1 using neq3 unfolding NEG-INF-def by auto
       then have sint((− 2147483647)::word) ≠ sint w1
        using word-sint.Rep-inject by blast
       then have n1:− 2147483647 ≠ sint w1
        by auto
       have − 2147483648 ≠ w1
        apply(rule repe.cases[OF equiv1]) using int-not-undef[of w1] by auto
       then have sint(− 2147483648::word) ≠ sint w1
        using word-sint.Rep-inject by blast
       then have n2:− 2147483648 ≠ sint w1
        by auto
       then have d:sint w1 > − 2147483647
```

```
        using word-sint.Rep[of (w1)::32 Word.word]
          sints32 len32[of TYPE(32)] n1 n2 neq3
        by (simp)
      have d2:sint (0x7FFFFFFF::64 Word.word) > 0
        by auto
       from d d2 have d3:- 2147483647 < sint w1 + sint (0x7FFFFFFF::64
Word.word)
        by auto
      have d4:sint ((scast (((scast w1)::64 Word.word) + 0x7FFFFFFF))::word)
        = sint w1 + sint (0x7FFFFFFF::64 Word.word)
        using case1a rightSize scast-down-range scast-eq1 by fastforce
       then show - 2147483647 < sint (SCAST(64 → 32) (SCAST(32 → 64)
w1 + 0x7FFFFFFF))
        using d3 d4 by auto
      qed done
    have (let sum = ?sum in if scast POS-INF <=s sum then POS-INF else scast
sum) ≡L r1 + r2
      using case1 case2
      by (auto simp add: Let-def)
     then show ?thesis
      using neq1 eq2 neq3 by (auto)
next
  case NumNum
  assume notinf1:w1 ≠ POS-INF
  assume notinf2:w2 ≠ POS-INF
  assume notneginf1:w1 ≠ NEG-INF
  assume notneginf2:w2 ≠ NEG-INF
  let ?sum = ((scast w1)::64 Word.word) + ((scast w2):: 64 Word.word)
  have truth: - (2 ^(size ((scast w1)::64 Word.word) - 1))
     ≤ sint ((scast w1)::64 Word.word) + sint ((scast w2)::64 Word.word)
   ∧ sint ((scast w1)::64 Word.word) + sint ((scast w2)::64 Word.word)
     ≤ 2 ^(size ((scast w1)::64 Word.word) - 1) - 1
   using Word.word-size[of (scast w2)::64 Word.word]
     Word.word-size[of (scast w1)::64 Word.word]
     scast-eq1 scast-eq3 sints64 sints32
     word-sint.Rep[of (w1)::32 Word.word]
     word-sint.Rep[of (w2)::32 Word.word]
   by auto
  have sint-eq:sint((scast w1 + scast w2)::64 Word.word) = sint w1 + sint w2
   using signed-arith-sint(1)[of (scast w1)::64 Word.word (scast w2)::64 Word.word,
OF truth]
     scast-eq1 scast-eq3
   by auto
  have bigOne:scast w1 + scast w2 <=s ((- 0x7FFFFFFF)::64 Word.word)
   ⟹ ∃ r'≤r1 + r2. r' ≤ -0x7FFFFFFF
  proof -
   assume scast w1 + scast w2 <=s ((- 0x7FFFFFFF)::64 Word.word)
   then have sum-leq:sint w1 + sint w2 ≤ - 0x7FFFFFFF
        and sum-leq': (sint w1 + sint w2) ≤ (- 2147483647)
```

using *sint-eq* **unfolding** *word-sle-eq* **by** *auto*
obtain $r'_1\ r'_2$ ::*real* **where**
bound1:$r'_1 \leq r1 \wedge (w1 \equiv_E r'_1)$ **and**
bound2:$r'_2 \leq r2 \wedge (w2 \equiv_E r'_2)$
using *lo1 lo2* **unfolding** *repL-def* **by** *auto*
have *somethingA*:$r'_1 \leq$ *sint w1* **and** *somethingB*:$r'_2 \leq$ *sint w2*
using *bound1 bound2* ‹*scast w1 + scast w2 <=s −0x7FFFFFFF*› *word-sle-def notinf1 notinf2*
unfolding *repe.simps* **by** *auto*
have *something*:$r'_1 + r'_2 \leq$ *sint w1 + sint w2*
using *somethingA somethingB add-mono*
by *fastforce*
show $\exists\, r' \leq r1 + r2.\ r' \leq (-0x7FFFFFFF)$
apply (*rule exI*[**where** $x = r'_1 + r'_2$])
using *bound1 bound2 add-mono something sum-leq'*
apply (*auto intro*: *order-trans* [*of* - ‹*signed-real-of-word w1 + signed-real-of-word w2*›])
done
qed
have *anImp*:$\bigwedge r'.\ (r' \geq r1 + r2 \wedge r' \leq (-\ 2147483647)) \Longrightarrow$
$(\exists\, r.\ -\ (2\ \hat{}\ 31\ -\ 1) = -\ (2\ \hat{}\ 31\ -\ 1) \wedge r' = r \wedge r$
$\leq$ (*real-of-int* (*sint* ((− (2 ^ 31 − 1))::*32 Word.word*))))
by *auto*
have *anEq*:((*scast* ((− (2 ^ 31 − 1))::*32 Word.word*))::*64 Word.word*) = (*−0x7FFFFFFF*)
by *auto*
have *bigTwo*:
$\neg(((scast\ POS\text{-}INF)::64\ Word.word) <=s\ ?sum) \Longrightarrow$
$\neg(?sum <=s ((scast\ NEG\text{-}INF)::64\ Word.word)) \Longrightarrow$
$\exists\, r' \leq r1 + r2.\ r' = (real\text{-}of\text{-}int\ (sint\ (scast\ (((scast\ w1)::64\ Word.word)$
$+ ((scast\ w2)::64\ Word.word))::word)))$
$\wedge\ (r' <\ 0x7FFFFFFF \wedge\ (-0x7FFFFFFF) < r')$
proof −
assume $\neg(((scast\ POS\text{-}INF)::64\ Word.word) <=s\ ?sum)$
then have *sum-leq*:*sint w1 + sint w2 < 0x7FFFFFFF*
unfolding *word-sle-eq* **using** *sint-eq* **by** *auto*
then have *sum-leq'*: (*sint w1 + sint w2*) < (*2147483647*)
by *auto*
assume $\neg(?sum <=s ((scast\ NEG\text{-}INF)::64\ Word.word))$
then have *sum-geq*:(− *0x7FFFFFFF*) < *sint w1 + sint w2*
unfolding *word-sle-eq* **using** *sint-eq* **by** *auto*
then have *sum-geq'*: (− *2147483647*) < (*sint w1 + sint w2*)
by *auto*
obtain $r'_1\ r'_2$ ::*real* **where**
bound1:$r'_1 \leq r1 \wedge (w1 \equiv_E r'_1)$ **and**
bound2:$r'_2 \leq r2 \wedge (w2 \equiv_E r'_2)$
using *lo1 lo2* **unfolding** *repL-def* **by** *auto*
have *somethingA*:$r'_1 \leq$ *sint w1* **and** *somethingB*:$r'_2 \leq$ *sint w2*
using *word-sle-def notinf1 notinf2 bound1 bound2* **unfolding** *repe.simps* **by**

```
auto
    have something:r'_1 + r'_2 ≤  sint w1 + sint w2
      using somethingA somethingB add-mono
      by fastforce
    have (w1 ≡_E r'_1) using bound1 by auto
    then have
          r1w1:r'_1 = (real-of-int (sint w1))
      and w1U:(real-of-int (sint w1)) < (real-of-int (sint POS-INF))
      and w1L:(real-of-int (sint NEG-INF)) < (real-of-int (sint w1))
      unfolding repe.simps
      using notinf1 notinf2 notneginf1 notneginf2 by (auto)
    have (w2 ≡_E r'_2) using bound2 by auto
    then have
          r2w1:r'_2 =  (real-of-int (sint w2))
      and w2U:(real-of-int (sint w2)) < (real-of-int (sint POS-INF))
      and w2L:(real-of-int (sint NEG-INF)) < (real-of-int (sint w2))
      unfolding repe.simps
      using notinf1 notinf2 notneginf1 notneginf2 by (auto)
    have sint (((scast w1)::64 Word.word) + ((scast w2)::64 Word.word))
     = sint ((scast (((scast w1)::64 Word.word) + ((scast w2)::64 Word.word)))::word)
      apply(rule scast-down-range)
      unfolding sint-eq using sints32 sum-geq sum-leq by auto
    then have cast-eq:(sint ((scast (((scast w1)::64 Word.word)
       + ((scast w2)::64 Word.word)))::word))
      = sint w1 + sint w2
      using scast-down-range sints32 sum-geq sum-leq sint-eq by auto
    from something and cast-eq
    have r12-sint-scast:r'_1 + r'_2
    = (sint ((scast (((scast w1)::64 Word.word) + ((scast w2)::64 Word.word)))::word))
      using r1w1 w1U w1L r2w1 w2U w2L
      by (simp)
    have leq-ref:⋀x y ::real. x = y ==> x ≤ y by auto
    show ?thesis
      apply(rule exI[where x=r'_1 + r'_2])
      apply(rule conjI)
      subgoal
        using r12-sint-scast cast-eq leq-ref r2w1 r1w1 add-mono[of r'_1 r1 r'_2 r2]
bound1 bound2
        by auto
     using bound1 bound2 add-mono r12-sint-scast cast-eq sum-leq sum-leq' sum-geq'
sum-geq
      ‹r'_1 + r'_2 =  (real-of-int (sint (scast (scast w1 + scast w2))))›
      by auto
    qed
  have neg-inf-case:?sum <=s ((scast ((NEG-INF)::word))::64 Word.word)  ⟹
NEG-INF ≡_L r1 + r2
  proof (unfold repL-def NEG-INF-def repe.simps)
     assume scast w1 + scast w2 <=s ((scast ((- (2 ^ 31 - 1))::word))::64
Word.word)
```

then have *scast w1 + scast w2 <=s ((− 0x7FFFFFFF)::64 Word.word)*
by (*metis anEq*)
then obtain r' **where** *geq*:($r' \leq r1 + r2$) **and** *leq*:($r' \leq (-0x7FFFFFFF)$)
using *bigOne* **by** *auto*
show ($\exists\, r' \leq$*plus r1 r2*.
($\exists\, r$. *uminus* (*minus*(*2 ^ 31*) *1*) = *POS-INF* $\wedge$ $r' = r \wedge$ (*real-of-int* (*sint POS-INF*)) $\leq r$)
$\vee$ ($\exists\, r$. *uminus* (*minus*(*2 ^ 31*) *1*) = *uminus* (*minus*(*2 ^ 31*) *1*)
$\wedge\ r' = r \wedge r \leq$ (*real-of-int* (*sint* ((*uminus* (*minus*(*2 ^ 31*) *1*))::*word*))))
$\vee$ ($\exists\, w$. *uminus* (*minus*(*2 ^ 31*) *1*) = *w* $\wedge$
$r' =$ (*real-of-int* (*sint w*)) $\wedge$
(*real-of-int* (*sint w*)) < (*real-of-int* (*sint POS-INF*))
$\wedge$ *less* ((*real-of-int* (*sint* (*uminus* (*minus*(*2 ^ 31*) *1*))))) ((*real-of-int* (*sint w*)))))
using *leq geq*
by (*meson dual-order.strict-trans linorder-not-le order-less-irrefl*)
qed
have *bigThree*:*0x7FFFFFFF <=s* ((*scast w1*)::*64 Word.word*) + ((*scast w2*)::*64 Word.word*)
$\Longrightarrow \exists\, r' \leq$*r1 + r2*. *2147483647* $\leq r'$
proof −
assume *0x7FFFFFFF <=s* ((*scast w1*)::*64 Word.word*) + ((*scast w2*)::*64 Word.word*)
then have *sum-leq*:*0x7FFFFFFF* $\leq$ *sint w1 + sint w2*
and *sum-leq'*: *2147483647* $\leq$ (*sint w1 + sint w2*)
using *sint-eq* **unfolding** *word-sle-eq* **by** *auto*
obtain $r'_1\ r'_2$ ::*real* **where**
bound1:$r'_1 \leq r1 \wedge (w1 \equiv_E r'_1)$ **and**
bound2:$r'_2 \leq r2 \wedge (w2 \equiv_E r'_2)$
using *lo1 lo2* **unfolding** *repL-def* **by** *auto*
have *somethingA*:$r'_1 \leq$ *sint w1* **and** *somethingB*:$r'_2 \leq$ *sint w2*
using ‹ *0x7FFFFFFF <=s scast w1 + scast w2* › *word-sle-def notinf1 notinf2 bound1 bound2 repe.simps*
by *auto*
have *something*:$r'_1 + r'_2 \leq$ *sint w1 + sint w2*
using *somethingA somethingB add-mono of-int-add*
by *fastforce*
show $\exists\, r' \leq$ *r1 + r2*. (*2147483647*) $\leq r'$
apply(*rule exI*[**where** $x = r'_1 + r'_2$])
using *bound1 bound2 add-mono something sum-leq' order.trans*
proof −
have *f1*: (*real-of-int* (*sint w2*)) = r'_2
by (*metis bound2 notinf2 notneginf2 repe.cases*)
have (*real-of-int* (*sint w1*)) = r'_1
by (*metis bound1 notinf1 notneginf1 repe.cases*)
then have *f2*: (*real-of-int 2147483647*) $\leq r'_2 + r'_1$
using *f1 sum-leq'* **by** *auto*
have $r'_2 + r'_1 \leq$ *r2 + r1*
by (*meson add-left-mono add-right-mono bound1 bound2 order.trans*)

```
    then show r'1 + r'2 ≤ r1 + r2 ∧  2147483647 ≤ r'1 + r'2
      using f2 by (simp add: add.commute)
   qed
  qed
  have inf-case:((scast POS-INF)::64 Word.word) <=s ?sum ⟹ POS-INF ≡L
r1 + r2
   proof −
    assume ((scast POS-INF)::64 Word.word)
       <=s ((scast w1)::64 Word.word) + ((scast w2)::64 Word.word)
    then have ((scast ((2 ^ 31 − 1)::word))::64 Word.word)
         <=s ((scast w1)::64 Word.word) + ((scast w2)::64 Word.word)
      unfolding repL-def repe.simps by auto
    then have (0x7FFFFFFF::64 Word.word)
         <=s ((scast w1)::64 Word.word) + ((scast w2)::64 Word.word)
      by auto
    then obtain r' where geq:(r' ≤ r1 + r2) and leq:(0x7FFFFFFF ≤ r')
      using bigThree by auto
    show ?thesis
      unfolding repL-def repe.simps using leq geq by auto
   qed
  have int-case:¬(((scast POS-INF)::64 Word.word) <=s ?sum)
   ⟹ ¬ (?sum <=s ((scast NEG-INF)::64 Word.word))
   ⟹ ((scast ?sum)::word) ≡L r1 + r2
  proof −
   assume bound1:¬ ((scast POS-INF)::64 Word.word) <=s scast w1 + scast w2
   assume bound2:¬ scast w1 + scast w2 <=s ((scast NEG-INF)::64 Word.word)
   obtain r'::real
    where rDef:r' = (real-of-int (sint ((scast (((scast w1)::64 Word.word)
                + ((scast w2)::64 Word.word)))::word)))
    and r12:r'≤r1 + r2
    and boundU:r' <  0x7FFFFFFF
    and boundL: (−0x7FFFFFFF) < r'
    using bigTwo[OF bound1 bound2] by auto
   obtain w::word
  where wdef:w = (scast (((scast w1)::64 Word.word) + ((scast w2)::64 Word.word))::word)
    by auto
   then have wr:r' =  (real-of-int (sint w))
    using r12 bound1 bound2 rDef by blast
   show ?thesis
    unfolding repL-def repe.simps
    using r12 wdef rDef boundU boundL wr
    by auto
  qed
  have (let sum = ?sum in
   if scast POS-INF <=s sum then POS-INF
   else if sum <=s scast NEG-INF then NEG-INF
   else scast sum) ≡L r1 + r2
   apply(cases ((scast POS-INF)::64 Word.word) <=s ?sum)
    apply(cases ?sum <=s scast NEG-INF)
```

using *inf-case neg-inf-case int-case* **by** (*auto simp add*: *Let-def*)
then show *?thesis*
using *notinf1 notinf2 notneginf1 notneginf2*
by(*auto*)
qed
qed

3.3 Max function

Maximum of w1 + w2 in 2s-complement

fun *wmax* :: *word* ⇒ *word* ⇒ *word*
where *wmax w1 w2* = (*if w1* <*s w2 then w2 else w1*)

Correctness of wmax

lemma *wmax-lemma*:
assumes *eq1*:*w1* ≡$_E$ (*r1*::*real*)
assumes *eq2*:*w2* ≡$_E$ (*r2*::*real*)
shows *wmax w1 w2* ≡$_E$ (*max r1 r2*)
proof(*cases rule*: *case-inf2*[**where** *?w1.0*=*w1*, **where** *?w2.0*=*w2*])
case *PosPos*
from *PosPos eq1 eq2*
have *bound1*:(*real-of-int* (*sint POS-INF*)) ≤ *r1*
and *bound2*:(*real-of-int* (*sint POS-INF*)) ≤ *r2*
by (*auto simp add*: *repe.simps*)
have *eqInf*:*wmax w1 w2* = *POS-INF*
using *PosPos* **unfolding** *wmax.simps* **by** *auto*
have *pos-eq*:*POS-INF* ≡$_E$ *max r1 r2*
apply(*rule repPOS-INF*)
using *bound1 bound2*
by *linarith*
show *?thesis*
using *pos-eq eqInf* **by** *auto*
next
case *PosNeg*
from *PosNeg*
have *bound1*:(*real-of-int* (*sint POS-INF*)) ≤ *r1*
and *bound2*:*r2* ≤ (*real-of-int* (*sint NEG-INF*))
using *eq1 eq2* **by** (*auto simp add*: *repe.simps*)
have *eqNeg*:*wmax w1 w2* = *POS-INF*
unfolding *eq1 eq2 wmax.simps PosNeg word-sless-def word-sle-def*
by(*auto*)
have *neg-eq*:*POS-INF* ≡$_E$ *max r1 r2*
apply(*rule repPOS-INF*)
using *bound1 bound2 eq1 eq2* **by** *auto*
show *?thesis*
using *eqNeg neg-eq* **by** *auto*
next
case *PosNum*
from *PosNum eq1 eq2*

```
  have bound1: (real-of-int (sint POS-INF)) ≤ r1
  and bound2a: (real-of-int (sint NEG-INF)) < (real-of-int (sint w2))
  and bound2b: (real-of-int (sint w2)) < (real-of-int (sint POS-INF))
    by (auto simp add: repe.simps)
  have eqNeg:wmax w1 w2 = POS-INF
    using PosNum bound2b
    unfolding wmax.simps word-sless-def word-sle-def
    by auto
  have neg-eq:POS-INF ≡E max r1 r2
    apply (rule repPOS-INF)
    using bound1 bound2a bound2b word-sless-alt le-max-iff-disj
    unfolding eq1 eq2 by blast
  show ?thesis
    using eqNeg neg-eq by auto
next
  case NegPos
  from NegPos eq1 eq2
  have bound1:r1 ≤ (real-of-int (sint NEG-INF))
   and bound2: (real-of-int (sint POS-INF)) ≤ r2
    by (auto simp add: repe.simps)
  have eqNeg:wmax w1 w2 = POS-INF
    unfolding NegPos word-sless-def word-sle-def
    by(auto)
  have neg-eq:POS-INF ≡E max r1 r2
    apply(rule repPOS-INF)
    using bound1 bound2 by auto
  show wmax w1 w2 ≡E max r1 r2
    using eqNeg neg-eq by auto
next
  case NegNeg
  from NegNeg eq1 eq2
  have bound1:r1 ≤ (real-of-int (sint NEG-INF))
   and bound2:r2 ≤ (real-of-int (sint NEG-INF))
    by (auto simp add: repe.simps)
  have eqNeg:NEG-INF ≡E max r1 r2
    apply(rule repNEG-INF)
    using eq1 eq2 bound1 bound2
    by(auto)
  have neg-eq:wmax w1 w2 = NEG-INF
    using NegNeg by auto
  show wmax w1 w2 ≡E max r1 r2
    using eqNeg neg-eq by auto
next
  case NegNum
  from NegNum eq1 eq2
  have eq3:r2 = (real-of-int (sint w2))
   and bound2a:(real-of-int (sint w2)) < (real-of-int (sint POS-INF))
   and bound2b:(real-of-int (sint NEG-INF)) < (real-of-int (sint w2))
   and bound1:r1 ≤ (real-of-int (sint NEG-INF))
```

```
    by (auto simp add: repe.simps)
  have eqNeg:max r1 r2 = (real-of-int (sint w2))
    using NegNum assms(2) bound2a eq3 repeInt-simps bound1 bound2a bound2b
    by (metis less-irrefl max.bounded-iff max-def not-less)
  then have extra-eq:(wmax w1 w2) = w2
    using assms(2) bound2a eq3 NegNum repeInt-simps
    by (simp add: word-sless-alt)
  have neg-eq:wmax w1 w2 ≡E (real-of-int (sint (wmax w1 w2)))
    apply(rule repINT)
    using extra-eq eq3 bound2a bound2b by(auto)
  show wmax w1 w2 ≡E max r1 r2
    using eqNeg neg-eq extra-eq by auto
next
  case NumPos
  from NumPos eq1 eq2
  have p2:w2 = POS-INF
   and eq1:r1 = (real-of-int (sint w1))
   and eq2:r2 = r2
   and bound1a:(real-of-int (sint w1)) < (real-of-int (sint POS-INF))
   and bound1b:(real-of-int (sint NEG-INF)) < (real-of-int (sint w1))
   and bound2:(real-of-int (sint POS-INF)) ≤ r2
    by (auto simp add: repe.simps)
  have res1:wmax w1 w2 = POS-INF
    using NumPos p2 eq1 eq2 assms(1) bound1b p2 repeInt-simps
    by (simp add: word-sless-alt)
  have res3:POS-INF ≡E max r1 r2
    using repPOS-INF NumPos bound2 le-max-iff-disj by blast
  show wmax w1 w2 ≡E max r1 r2
    using res1 res3 by auto
next
  case NumNeg
  from NumNeg eq1 eq2
  have n2:w2 = NEG-INF
   and rw1:r1 = (real-of-int (sint w1))
   and bound1a:(real-of-int (sint w1)) < (real-of-int (sint POS-INF))
   and bound1b:(real-of-int (sint NEG-INF)) < (real-of-int (sint w1))
   and bound2:r2 ≤ (real-of-int (sint NEG-INF))
    by (auto simp add: repe.simps)
  have res1:max r1 r2 = (real-of-int (sint (wmax w1 w2)))
    using bound1b bound2 NumNeg less-trans wmax.simps of-int-less-iff
      word-sless-alt rw1 antisym-conv2 less-imp-le max-def
    by metis
  have res2:wmax w1 w2 ≡E (real-of-int (sint (wmax w1 w2)))
    apply(rule repINT)
    using bound1a bound1b bound2 NumNeg leD leI less-trans n2 wmax.simps
    by (auto simp add: word-sless-alt)
  show wmax w1 w2 ≡E max r1 r2
    using res1 res2 by auto
next
```

case *NumNum*
from *NumNum eq1 eq2*
have *eq1*:*r1* = (*real-of-int* (*sint w1*))
and *eq2*:*r2* = (*real-of-int* (*sint w2*))
and *bound1a*:(*real-of-int* (*sint w1*)) < (*real-of-int* (*sint POS-INF*))
and *bound1b*:(*real-of-int* (*sint NEG-INF*)) < (*real-of-int* (*sint w1*))
and *bound2a*:(*real-of-int* (*sint w2*)) < (*real-of-int* (*sint POS-INF*))
and *bound2b*:(*real-of-int* (*sint NEG-INF*)) < (*real-of-int* (*sint w2*))
by (*auto simp add*: *repe.simps*)
have *res1*:*max r1 r2* = (*real-of-int* (*sint* (*wmax w1 w2*)))
using *eq1 eq2 bound1a bound1b bound2a bound2b*
apply (*auto simp add*: *max-def word-sless-alt not-less*; *transfer*)
apply *simp-all*
done
have *res2*:*wmax w1 w2* ≡$_E$ (*real-of-int* (*sint* (*wmax w1 w2*)))
apply (*rule repINT*)
using *bound1a bound1b bound2a bound2b*
by (*simp add*: ‹*max r1 r2* = (*real-of-int* (*sint* (*wmax w1 w2*)))› *eq2 min-less-iff-disj*)+
show *wmax w1 w2* ≡$_E$ *max r1 r2*
using *res1 res2* **by** *auto*
qed

lemma *max-repU1*:
assumes *w1* ≡$_U$ *x*
assumes *w2* ≡$_U$ *y*
shows *wmax w1 w2* ≡$_U$ *x*
using *wmax-lemma assms repU-def*
by (*meson le-max-iff-disj*)

lemma *max-repU2*:
assumes *w1* ≡$_U$ *y*
assumes *w2* ≡$_U$ *x*
shows *wmax w1 w2* ≡$_U$ *x*
using *wmax-lemma assms repU-def*
by (*meson le-max-iff-disj*)

Product of w1 * w2 with bounds checking

fun *wtimes* :: *word* ⇒ *word* ⇒ *word*
where *wtimes w1 w2* =
(*if w1* = *POS-INF* ∧ *w2* = *POS-INF then POS-INF*
else if w1 = *NEG-INF* ∧ *w2* = *POS-INF then NEG-INF*
else if w1 = *POS-INF* ∧ *w2* = *NEG-INF then NEG-INF*
else if w1 = *NEG-INF* ∧ *w2* = *NEG-INF then POS-INF*

else if w1 = *POS-INF* ∧ *w2* <*s* *0 then NEG-INF*
else if w1 = *POS-INF* ∧ *0* <*s* *w2 then POS-INF*
else if w1 = *POS-INF* ∧ *0* = *w2 then 0*
else if w1 = *NEG-INF* ∧ *w2* <*s* *0 then POS-INF*
else if w1 = *NEG-INF* ∧ *0* <*s* *w2 then NEG-INF*

else if w1 = NEG-INF ∧ 0 = w2 then 0

else if w1 <s 0 ∧ w2 = POS-INF then NEG-INF
else if 0 <s w1 ∧ w2 = POS-INF then POS-INF
else if 0 = w1 ∧ w2 = POS-INF then 0
else if w1 <s 0 ∧ w2 = NEG-INF then POS-INF
else if 0 <s w1 ∧ w2 = NEG-INF then NEG-INF
else if 0 = w1 ∧ w2 = NEG-INF then 0

else
*(let prod::64 Word.word = (scast w1) * (scast w2) in*
if prod <=s (scast NEG-INF) then NEG-INF
else if (scast POS-INF) <=s prod then POS-INF
else (scast prod)))

3.4 Multiplication upper bound

Product of 32-bit numbers fits in 64 bits

lemma *times-upcast-lower*:
fixes *x y*::*int*
assumes *a1*:$x \geq -2147483648$
assumes *a2*:$y \geq -2147483648$
assumes *a3*:$x \leq 2147483648$
assumes *a4*:$y \leq 2147483648$
shows $-4611686018427387904 \leq x * y$
proof −
let *?thesis* = $-4611686018427387904 \leq x * y$
have *is-neg*:$-4611686018427387904 < (0::int)$ **by** *auto*
have *case1*:$x \geq 0 \Longrightarrow y \geq 0 \Longrightarrow$ *?thesis*
proof −
assume *a5*:$x \geq 0$ **and** *a6*:$y \geq 0$
have *h1*:$x * y \geq 0$
using *a5 a6* **by** (*simp add*: *zero-le-mult-iff*)
then show *?thesis* **using** *is-neg* **by** *auto*
qed
have *case2*:$x \leq 0 \Longrightarrow y \geq 0 \Longrightarrow$ *?thesis*
proof −
assume *a5*:$x \leq 0$ **and** *a6*:$y \geq 0$
have *h1*:$-2147483648 * (2147483648) \leq x * 2147483648$
using *a1 a2 a3 a4 a5 a6* **by** *linarith*
have *h2*:$-2147483648 \leq y$ **using** *a6* **by** *auto*
have *h3*:$x * 2147483648 \leq x * y$
using *a1 a2 a3 a4 a5 a6 h2*
using *mult-left-mono-neg* **by** *blast*
show *?thesis* **using** *h1 h2 h3* **by** *auto*
qed
have *case3*:$x \geq 0 \Longrightarrow y \leq 0 \Longrightarrow$ *?thesis*
proof −
assume *a5*:$x \geq 0$ **and** *a6*:$y \leq 0$

```
    have h1:2147483648 * (-2147483648) ≤ 2147483648 * y
      using a1 a2 a3 a4 a5 a6 by linarith
    have h2:-2147483648 ≤ x using a5 by auto
    have h3:2147483648 * y ≤ x * y
      using a1 a2 a3 a4 a5 a6 h2
      using mult-left-mono-neg mult-right-mono-neg by blast
    show ?thesis using h1 h2 h3 by auto
  qed
 have case4:x ≤ 0 ⟹ y ≤ 0 ⟹ ?thesis
 proof -
    assume a5:x ≤ 0 and a6:y ≤ 0
    have h1:x * y ≥ 0
      using a5 a6 by (simp add: zero-le-mult-iff)
    then show ?thesis using is-neg by auto
  qed
 show ?thesis
  using case1 case2 case3 case4 by linarith
qed
```

Product of 32-bit numbers fits in 64 bits

```
lemma times-upcast-upper:
 fixes x y ::int
 assumes a1:x ≥ -2147483648
 assumes a2:y ≥ -2147483648
 assumes a3:x ≤ 2147483648
 assumes a4:y ≤ 2147483648
 shows x * y ≤ 4611686018427387904
proof -
 let ?thesis = x * y ≤ 4611686018427387904
 have case1:x ≥ 0 ⟹ y ≥ 0 ⟹ ?thesis
  proof -
    assume a5:x ≥ 0 and a6:y ≥ 0
    have h1:2147483648 * 2147483648 ≥ x * 2147483648
      using a1 a2 a3 a4 a5 a6 by linarith
    have h2:x * 2147483648 ≥ x * y
      using a1 a2 a3 a4 a5 a6 by (simp add: mult-mono)
    show ?thesis using h1 h2 by auto
  qed
 have case2:x ≤ 0 ⟹ y ≥ 0 ⟹ ?thesis
  proof -
    assume a5:x ≤ 0 and a6:y ≥ 0
    have h1:2147483648 * 2147483648 ≥ (0::int) by auto
    have h2:0 ≥ x * y
      using a5 a6 mult-nonneg-nonpos2 by blast
    show ?thesis using h1 h2 by auto
  qed
 have case3:x ≥ 0 ⟹ y ≤ 0 ⟹ ?thesis
  proof -
    assume a5:x ≥ 0 and a6:y ≤ 0
```

```
    have h1:2147483648 * 2147483648 ≥ (0::int) by auto
    have h2:0 ≥ x * y
      using a5 a6 mult-nonneg-nonpos by blast
    show ?thesis using h1 h2 by auto
   qed
  have case4:x ≤ 0 ⟹ y ≤ 0 ⟹ ?thesis
   proof −
    assume a5:x ≤ 0 and a6:y ≤ 0
    have h1:−2147483648 * −2147483648 ≥ x * −2147483648
      using a1 a2 a3 a4 a5 a6 by linarith
    have h2:x * −2147483648 ≥ x * y
      using a1 a2 a3 a4 a5 a6 mult-left-mono-neg by blast
    show ?thesis using h1 h2 by auto
   qed
show x * y ≤ 461168601842738790 4
  using case1 case2 case3 case4
  by linarith
qed
```

Correctness of 32x32 bit multiplication

3.5 Exact multiplication

```
lemma wtimes-exact:
assumes eq1:w1 ≡E r1
assumes eq2:w2 ≡E r2
shows wtimes w1 w2 ≡E r1 * r2
proof −
  have POS-cast:sint ((scast POS-INF)::64 Word.word) = sint POS-INF
    apply(rule Word.sint-up-scast)
    unfolding Word.is-up by auto
  have POS-sint:sint POS-INF = (2^31)−1  by auto
  have w1-cast:sint ((scast w1)::64 Word.word) = sint w1
    apply(rule Word.sint-up-scast)
    unfolding Word.is-up by auto
  have w2-cast:sint ((scast w2)::64 Word.word) = sint w2
    apply(rule Word.sint-up-scast)
    unfolding Word.is-up by auto
  have NEG-cast:sint ((scast NEG-INF)::64 Word.word) = sint NEG-INF
    apply(rule Word.sint-up-scast)
    unfolding Word.is-up by auto
  have rangew1:sint ((scast w1)::64 Word.word) ∈ {− (2 ^ 31).. (2^31)}
     using word-sint.Rep[of (w1)::32 Word.word] sints32 len32 mem-Collect-eq
POS-cast w1-cast
    by auto
  have rangew2:sint ((scast w2)::64 Word.word) ∈ {− (2 ^ 31).. (2^31)}
     using word-sint.Rep[of (w2)::32 Word.word] sints32 len32 mem-Collect-eq
POS-cast w2-cast
    by auto
```

show *?thesis*
proof (*cases rule*: *case-times-inf*[*of w1 w2*])
case *PosPos* **then**
have *a1*: *PosInf* ≤ *r1*
and *a2*: *PosInf* ≤ *r2*
using *PosPos eq1 eq2 repe.simps* **by** (*auto*)
have *f3*: ⋀*n e*::*real*. *1* ≤ *max* ((*numeral n*)) *e*
by (*simp add*: *le-max-iff-disj*)
have ⋀*n e*::*real*. *0* ≤ *max* ((*numeral n*)) *e*
by (*simp add*: *le-max-iff-disj*)
then have *r1* ≤ *r1* ∗ *r2*
using *f3 PosPos eq1 eq2 repe.simps*
using *eq1 eq2* **by** (*auto simp add*: *repe.simps*)
then have *PosInf* ≤ *r1* ∗ *r2*
using *a1* **by** *linarith*
then show *?thesis*
using *PosPos* **by**(*auto simp add*: *repe.simps*)
next
case *NegPos*
from *NegPos* **have** *notPos*:*w1* ≠ *POS-INF* **unfolding** *POS-INF-def NEG-INF-def*
by *auto*
have *a1*: *NegInf* ≥ *r1*
using *eq1 NegPos* **by** (*auto simp add*: *repe.simps*)
have *a2*: *PosInf* ≤ *r2*
using *eq2 NegPos* **by** (*auto simp add*: *repe.simps*)
have *f1*: *real-of-int Numeral1* = *1*
by *simp*
have *f3*: (*real-of-int 3*) ≤ − *r1*
using *a1* **by** (*auto*)
have *f4*:*0* ≤ *r2*
using *f1 a2* **by**(*auto*)
have *f5*: *r1* ≤ − *1*
using *f3* **by** *auto*
have *fact*:*r1* ∗ *r2* ≤ − *r2*
using *f5 f4 mult-right-mono* **by** *fastforce*
show *?thesis*
using *a1 a2 fact* **by** (*auto simp add*: *repe.simps NegPos*)
next
case *PosNeg*
have *a1*: *PosInf* ≤ *r1*
using *eq1 PosNeg* **by** (*auto simp add*: *repe.simps*)
then have *h1*:*r1* ≥ *1*
by (*auto*)
have *a2*: *NegInf* ≥ *r2*
using *eq2 PosNeg* **by** (*auto simp add*: *repe.simps*)
have *f1*: ¬ *NegInf* ∗ (− *1*) ≤ *1*
by (*auto*)
have *f2*: ∀ *e ea*::*real*. (*e* ∗ (− *1*) ≤ *ea*) = (*ea* ∗ (− *1*) ≤ *e*) **by** *force*
then have *f3*: ¬ *1* ∗ (− *1*::*real*) ≤ *NegInf*

```
    using f1 by blast
  have f4: r1 * (- 1) ≤ NegInf
    using f2 a1
    by(auto)
  have f5: ∀ e ea eb. (if (ea::real) ≤ eb then e ≤ eb else e ≤ ea) = (e ≤ ea ∨ e ≤
eb)
    by force
  have 0 * (- 1::real) ≤ 1
    by simp
  then have r1 * (- 1) ≤ 0
    using f5 f4 f3 f2 by meson
  then have f6: 0 ≤ r1 by fastforce
  have 1 * (- 1) ≤ (- 1::real)
    using f2 by force
  then have fact:r2 ≤ (- 1)
    using f3 a2 by fastforce
  have rule:⋀c. c > 0 ⟹ r1 ≥ c ∧ r2 ≤ -1 ⟹ r1 * r2 ≤ -c
    apply auto
    by (metis (no-types, opaque-lifting) f5 mult-less-cancel-left-pos
        mult-minus1-right neg-le-iff-le not-less)
  have r1 * r2 ≤ NegInf
    using PosNeg f6 fact rule[of PosInf] a1
    by(auto)
  then show ?thesis
    using PosNeg by (auto simp add: repe.simps)
next
  case NegNeg
  have a1: (-2147483647) ≥ r1
    using eq1 NegNeg by (auto simp add: repe.simps)
  then have h1:r1 ≤ -1
    using max.bounded-iff max-def one-le-numeral
    by auto
  have a2: (-2147483647) ≥ r2
    using eq2 NegNeg by (auto simp add: repe.simps)
  have f1: ∀ e ea eb. ¬ (e::real) ≤ ea ∨ ¬ 0 ≤ eb ∨ eb * e ≤ eb * ea
    using mult-left-mono by metis
  have f2: - 1 = (- 1::real)
    by force
  have f3: 0 ≤ (1::real)
    by simp
  have f4: ∀ e ea eb. (ea::real) ≤ e ∨ ¬ ea ≤ eb ∨ ¬ eb ≤ e
    by (meson less-le-trans not-le)
  have f5: 0 ≤ (2147483647::real)
    by simp
  have f6: - (- 2147483647) = (2147483647::real)
    by force
  then have f7: - ( (- 2147483647) * r1) = (2147483647 * r1)
    by (metis mult-minus-left)
  have f8: - ( (- 2147483647) * (- 1)) = 2147483647 * (- 1::real)
```

```
    by simp
  have  2147483647 = -  1 *  (- 2147483647::real)
    by simp
  then have f9: r1 ≤  (- 1) ⟶  2147483647 ≤ r1 *  (- 2147483647)
    using f8 f7 f5 f2 f1 by linarith
  have f10: -  2147483647 =  (- 2147483647::real)
    by fastforce
  have f11: - (r2 *  1 * (r1 *  (- 1))) = r1 * r2
    by (simp add: mult.commute)
  have f12: r1 *  (- 1) = - (r1 *  1)
    by simp
  have r1 *  1 * ( (- 2147483647) *  1) =  (- 2147483647) * r1
    by (simp add: mult.commute)
  then have f13: r1 *  (- 1) * ( (- 2147483647) *  1) =  2147483647 * r1
    using f12 f6 by (metis (no-types) mult-minus-left)
  have  1 * r1 ≤  1 *  (- 2147483647)
    using a1
    by (auto simp add: a1)
  then have  2147483647 ≤ r1 *  (- 1) by fastforce
  then have 0 ≤ r1 *  (- 1)
    using f5 f4 by (metis)
  then have r1 ≤  (- 1) ∧ - (r1 *  2147483647) ≤ - (r2 *  1 * (r1 *  (- 1)))
     by (metis a2 f11 h1 mult-left-mono-neg minus-mult-right mult-minus1-right
neg-0-le-iff-le)
  then have r1 ≤  (- 1) ∧ r1 *  (- 2147483647) ≤ r2 * r1
    using f11 f10 by (metis mult-minus-left mult.commute)
  then have fact: 2147483647 ≤ r2 * r1
    using f9 f4 by blast
  show ?thesis
    using a1 a2 h1 fact
    by (auto simp add: repe.simps NegNeg mult.commute)
next
  case PosLo
  from PosLo
  have w2NotPinf:w2 ≠ POS-INF and w2NotNinf:w2 ≠ NEG-INF by (auto)
  from eq1 PosLo
  have upper: (real-of-int (sint POS-INF)) ≤ r1
    by (auto simp add: repe.simps)
  have lower1:sint w2 < 0 using PosLo
    apply (auto simp add: word-sless-def word-sle-def)
    by (simp add: dual-order.order-iff-strict)
  then have lower2:sint w2 ≤ -1 by auto
  from eq2 have rw2:r2 =  (real-of-int (sint w2))
    using repe.simps PosLo
    by (auto simp add: repe.simps)
  have f4: r1 *  (- 1) ≤  (- 2147483647)
    using upper by (auto)
  have f5: r2 ≤  (- 1)
    using lower2 rw2 by transfer simp
```

```
  have 0 < r1
    using upper by (auto)
  have ∀ r. r < - 2147483647 ∨ ¬ r < r1 * - 1
    using f4 less-le-trans by blast
  then have r1 *  (real-of-int (sint w2)) ≤  (- 2147483647)
    using f5 f4 upper lower2 rw2 mult-left-mono
    by (metis ‹0 < r1› dual-order.order-iff-strict f5 mult-left-mono rw2)
  then have r1 * r2 ≤ real-of-int (sint NEG-INF)
    using upper lower2  rw2
    by (auto)
  then show ?thesis
    using PosLo by (auto simp add: repe.simps)
next
  case PosHi
  from PosHi
  have w2NotPinf:w2 ≠ POS-INF and w2NotNinf:w2 ≠ NEG-INF
    by (auto)
  from eq1 PosHi
  have upper:(real-of-int (sint POS-INF)) ≤ r1
    by (auto simp add: repe.simps)
  have lower1:sint w2 > 0 using PosHi
    apply (auto simp add: word-sless-def word-sle-def)
    by (simp add: dual-order.order-iff-strict)
  then have lower2:sint w2 ≥ 1 by auto
  from eq2 have rw2:r2 =  (real-of-int (sint w2))
    using repe.simps PosHi
    by (auto)
  have 0 ≤ r1 using upper by (auto)
  then have r1 ≤ r1 * r2
     using rw2 lower2 by (metis (no-types)  mult-left-mono mult.right-neutral
of-int-1-le-iff)
  have PosInf ≤ r1 * r2
     using upper lower2 rw2
     apply (auto)
     using ‹0 ≤ r1› mult-numeral-1-right numeral-One of-int-1-le-iff zero-le-one
     apply simp
    using mult-mono [of 2147483647 r1 1 ‹signed-real-of-word (w2::32 Word.word)›]
     apply simp
     apply transfer
     apply simp
     done
  then show ?thesis
  using PosHi by (auto simp add: repe.simps)
next
  case PosZero
  from PosZero
  have w2NotPinf:w2 ≠ POS-INF and w2NotNinf:w2 ≠ NEG-INF
    by (auto)
  from eq1 PosZero
```

```
  have upper: (real-of-int (sint POS-INF)) ≤ r1
    by (auto simp add: repe.simps)
  have lower1:sint w2 = 0 using PosZero
    by (auto simp add: word-sless-def word-sle-def)
  from eq2 have rw2:r2 =  (real-of-int (sint w2))
    using repe.simps PosZero
    by auto
  have 0 = r1 * r2
    using PosZero rw2 by auto
  then show ?thesis
    using PosZero by (auto simp add: repe.simps)
next
  case NegHi
  have w2NotPinf:w2 ≠ POS-INF and w2NotNinf:w2 ≠ NEG-INF
    using NegHi by (auto)
  from eq1 NegHi
  have upper:(real-of-int (sint NEG-INF)) ≥ r1
    by (auto simp add: repe.simps)
  have low:sint w2 > 0 using NegHi
    apply (auto simp add: word-sless-def word-sle-def)
    by (simp add: dual-order.order-iff-strict)
  then have lower1:(real-of-int (sint w2)) > 0
    by transfer simp
  then have lower2:(real-of-int (sint w2)) ≥ 1
    using low by transfer simp

  from eq1 have rw1:r1 ≤ (real-of-int (sint w1))
    using repe.simps NegHi
    by (simp add: upper)
  from eq2 have rw2:r2 = (real-of-int (sint w2))
    using repe.simps NegHi
    by (auto)
  have mylem:⋀x y z::real. x ≤ −1 ⟹ y ≥ 1 ⟹ z ≤ −1 ⟹ x ≤ z ⟹  x * y
≤ z
    proof −
    fix x y z::real
    assume a1:x ≤ −1
    assume a2:y ≥ 1
    then have h1:−1 ≥ −y by auto
    assume a3:z ≤ −1
    then have a4:z < 0  by auto
    from a4 have h2:−z > 0  using leD leI by auto
    from a3 have h5:−z ≥ 1  by (simp add: leD leI)
    assume a5:x ≤ z
    then have h6:−x ≥ −z by auto
    have h3:−x * −z = x * z by auto
    show x * y ≤ z
     using a1 a2 a3 a5 a4 h1 h2 h3 h6 h5 a5 dual-order.trans leD mult.right-neutral
      by (metis dual-order.order-iff-strict mult-less-cancel-left2)
```

```
    qed
  have prereqs:r1 ≤ − 1 1
     ≤ (real-of-int (sint w2))  (− 2147483647::real) ≤ − 1  r1 ≤ (−2147483647)
    using rw1 rw2 NegHi lower2 by (auto simp add: word-sless-def word-sle-def)
  have r1 ∗ r2 ≤ real-of-int (sint NEG-INF)
    using upper lower1 lower2 rw1 rw2
    apply (auto simp add: word-sless-def word-sle-def)
    using mylem[of  r1  (real-of-int (sint w2))  (− 2147483647)] prereqs
    by auto
  then show ?thesis
    using NegHi by (auto simp add: repe.simps)
next
  case NegLo
  from NegLo
  have w2NotPinf:w2 ≠ POS-INF and w2NotNinf:w2 ≠ NEG-INF
    by (auto)
  from eq1 NegLo
  have upper:(real-of-int (sint NEG-INF)) ≥ r1
  by (auto simp add: repe.simps)
  have low:sint w2 < 0 using NegLo
    by (auto simp add: word-sless-def word-sle-def dual-order.order-iff-strict)
  then have lower1:(real-of-int (sint w2)) < 0
    by transfer simp
  from eq1 have rw1:r1 ≤  (real-of-int (sint w1))
    using repe.simps NegLo
    by (simp add: upper)
  from eq2 have rw2:r2 =  (real-of-int (sint w2))
    using repe.simps NegLo
    by (auto)
  have hom:(− 2147483647) = −(2147483647::real) by auto
  have mylem:⋀x y z::real. y < 0 ⟹   x ≤ y ⟹ z ≤ −1 ⟹ −y ≤ x ∗ z
    proof −
    fix x y z::real
    assume a1:y < 0
    assume a2:x ≤ y
    then have h1:−x ≥ −y by auto
    assume a3:z ≤ −1
    then have a4:z < 0
      by auto
    from a4 have h2:−z > 0 using leD leI by auto
    from a3 have h5:−z ≥ 1 by (simp add: leD leI)
    have h4:−x ∗ −z ≥ −y
      using  a1 a2 a3 a4 h1 h2 h5 dual-order.trans mult.right-neutral
      by (metis mult.commute neg-0-less-iff-less mult-le-cancel-right-pos)
    have h3:−x ∗ −z = x ∗ z by auto
    show − y ≤ x ∗ z
      using a1 a2 a3 a4 h1 h2 h3 h4 h5
      by simp
    qed
```

```
  have prereqs:− 2147483647 < (0::real)  r1 ≤ − 2147483647
    using rw1 rw2 NegLo by (auto simp add: word-sless-def word-sle-def)
  moreover have ‹sint w2 ≤ − 1›
    using low by simp
  then have ‹real-of-int (sint w2) ≤ real-of-int (− 1)›
    by (simp only: of-int-le-iff)
  then have ‹signed-real-of-word w2 ≤ − 1›
    by simp
  ultimately have 2147483647 ≤ r1 * r2
    using upper lower1 rw1 rw2
      mylem[of −2147483647 r1 (real-of-int (sint w2))]
    by (auto simp add: word-sless-def word-sle-def)
  then show ?thesis
    using NegLo by (auto simp add: repe.simps)
next
  case NegZero
  from NegZero
  have w2NotPinf:w2 ≠ POS-INF and w2NotNinf:w2 ≠ NEG-INF by (auto)
  from eq2 NegZero
  have r2 = 0
    using repe.simps NegZero
    by (auto)
  then show ?thesis
    using NegZero by (auto simp add: repe.simps)
next
  case LoPos
  from LoPos
  have w2NotPinf:w1 ≠ POS-INF and w2NotNinf:w1 ≠ NEG-INF
    by (auto)
  from eq2 LoPos
  have upper:(real-of-int (sint POS-INF)) ≤ r2
    by (auto simp add: repe.simps)
  have lower1:sint w1 < 0 using LoPos
    apply (auto simp add: word-sless-def word-sle-def)
    by (simp add: dual-order.order-iff-strict)
  then have lower2:sint w1 ≤ −1 by auto
  from eq1 have rw1:r1 = (real-of-int (sint w1))
    using repe.simps LoPos by (auto simp add: repe.simps)
  have f4: r2 * (− 1) ≤ (− 2147483647)
    using upper by(auto)
  have f5: r1 ≤ (− 1)
    using lower2 rw1 by transfer simp
  have 0 < r2
    using upper by(auto)
  then have r2 * r1 ≤ r2 * (− 1)
    by (metis dual-order.order-iff-strict mult-right-mono f5  mult.commute)
  then have r2 * r1 ≤ (− 2147483647)
    by (meson f4 less-le-trans not-le)
  then have (real-of-int (sint w1)) * r2 ≤ (− 2147483647)
```

```
    using f5 f4 rw1 less-le-trans not-le mult.commute rw1
    by (auto simp add: mult.commute)
  then have r1 * r2 ≤ NegInf
    using rw1
    by (auto)
  then show ?thesis
    using LoPos by (auto simp: repe.simps)
next
  case HiPos
  from HiPos
  have w2NotPinf:w1 ≠ POS-INF and w2NotNinf:w1 ≠ NEG-INF
    by (auto)
  from eq2 HiPos
  have upper:(real-of-int (sint POS-INF)) ≤ r2
    by (auto simp add: repe.simps)
  have lower1:sint w1 > 0 using HiPos
    by (auto simp add: word-sless-def word-sle-def dual-order.order-iff-strict)
  then have lower2:sint w1 ≥ 1 by auto
  from eq1 have rw2:r1 = (real-of-int (sint w1))
    using HiPos
    by (auto simp add: repe.simps)
  have 0 ≤ r2
    using upper by(auto)
  then have r2 ≤ r2 * r1
   using lower2 rw2 by (metis (no-types) mult-left-mono mult.right-neutral of-int-1-le-iff)
  have 2147483647 ≤ r1 * r2
    using upper lower2 rw2
    apply (simp add: word-sless-def word-sle-def)
    using mult-mono [of 1 ‹signed-real-of-word w1› 2147483647 r2]
    apply simp
    apply transfer
    apply simp
    done
  then show ?thesis
    using HiPos by (auto simp add: repe.simps)
next
  case ZeroPos
  from ZeroPos
  have w2NotPinf:w1 ≠ POS-INF and w2NotNinf:w1 ≠ NEG-INF
    by (auto)
  from eq2 ZeroPos
  have upper: (real-of-int (sint POS-INF)) ≤ r2
    by (auto simp add: repe.simps)
  have lower1:sint w1 = 0 using ZeroPos
    by (auto simp add: word-sless-def word-sle-def)
  from eq1 have rw2:r1 = (real-of-int (sint w1))
    using repe.simps ZeroPos
    by (auto)
  have r1 = 0 using lower1 rw2 by auto
```

```
  then show ?thesis
    using ZeroPos by (auto simp add: repe.simps)
next
  case ZeroNeg
  from ZeroNeg
  have w2NotPinf:w1 ≠ POS-INF and w2NotNinf:w1 ≠ NEG-INF
    by (auto)
  from eq2 ZeroNeg
  have upper:(real-of-int (sint NEG-INF)) ≥ r2
    by (auto simp add: repe.simps)
  have lower1:sint w1 = 0 using ZeroNeg
    by (auto simp add: word-sless-def word-sle-def)
  from eq1 have rw2:r1 =  (real-of-int (sint w1))
    using repe.simps ZeroNeg
    by (auto)
  have r1 = 0 using lower1 rw2 by auto
  then show ?thesis
    using ZeroNeg by (auto simp add: repe.simps)
next
  case LoNeg
  from LoNeg
  have w2NotPinf:w1 ≠ POS-INF and w2NotNinf:w1 ≠ NEG-INF
    by (auto)
  from eq2 LoNeg
  have upper: (real-of-int (sint NEG-INF)) ≥ r2
    by (auto simp add: repe.simps)
  have low:sint w1 < 0 using LoNeg
    apply (auto simp add: word-sless-def word-sle-def)
    by (simp add: dual-order.order-iff-strict)
  then have lower1:(real-of-int (sint w1)) < 0 by transfer simp
  from low have ‹sint w1 ≤ − 1›
    by simp
  then have lower2:(real-of-int (sint w1)) ≤ −1
    by transfer simp
  from eq1 have rw1:r2 ≤  (real-of-int (sint w2))
    using LoNeg upper by auto
  from eq1 have rw2:r1 =  (real-of-int (sint w1))
    using LoNeg by (auto simp add: upper repe.simps)
  have hom:(− 2147483647::real) = −(2147483647) by auto
  have mylem:⋀x y z::real. y < 0 ⟹   x ≤ y ⟹ z ≤ −1 ⟹ −y ≤ x ∗ z
    proof −
    fix x y z::real
    assume a1:y < 0
    assume a2:x ≤ y
    then have h1:−x ≥ −y by auto
    assume a3:z ≤ −1
    then have a4:z < 0 by auto
    from a4 have h2:−z > 0
      using leD leI by auto
```

```
    from a3 have h5:−z ≥ 1
      by (simp add:  leD leI)
    have h4:−x ∗ −z ≥ −y
     using  a1 a2 a3 a4 h1 h2 h5 dual-order.trans mult-left-mono mult.right-neutral
mult.commute
    by (metis dual-order.order-iff-strict mult-minus-right mult-zero-right neg-le-iff-le)
    have h3:−x ∗ −z = x ∗ z by auto
    show − y ≤ x ∗ z
      using a1 a2 a3 a4 h1 h2 h3 h4 h5
      by simp
    qed
  have prereqs:− 2147483647 < (0::real)  r2 ≤ − 2147483647  (real-of-int (sint
w1)) ≤ − 1
     using rw1 rw2 LoNeg lower2 by (auto simp add: word-sless-def word-sle-def
lower2)
  have 2147483647 ≤ r1 ∗ r2
    using upper lower1 lower2 rw1 rw2 mylem[of −2147483647 r2
        (real-of-int (sint w1))] prereqs
    by (auto simp add:word-sless-def word-sle-def mult.commute)
  then show ?thesis
    using LoNeg by (auto simp add: repe.simps)
next
  case HiNeg
  from HiNeg
  have w1NotPinf:w1 ≠ POS-INF and w1NotNinf:w1 ≠ NEG-INF
    by (auto)
  have upper: (real-of-int (sint NEG-INF)) ≥ r2
    using HiNeg eq2
    by (auto simp add: repe.simps )
  have low:sint w1 > 0 using HiNeg
    apply (auto simp add: word-sless-def word-sle-def)
    by (simp add: dual-order.order-iff-strict)
  then have lower1:(real-of-int (sint w1)) > 0 by transfer simp
  from low have ‹sint w1 ≥ 1›
    by simp
  then have lower2:(real-of-int (sint w1)) ≥ 1
    by transfer simp
  from eq2 have rw1:r2 ≤  (real-of-int (sint w2))
    using repe.simps HiNeg by (simp add: upper)
  from eq1 have rw2:r1 =  (real-of-int (sint w1))
    using repe.simps HiNeg
    by (auto)
  have mylem:⋀x y z::real. x ≤ −1 ⟹ y ≥ 1 ⟹ z ≤ −1 ⟹ x ≤ z ⟹  x ∗ y
≤ z
    proof −
      fix x y z::real
      assume a1:x ≤ −1
      assume a2:y ≥ 1
      then have h1:−1 ≥ −y by auto
```

```
    assume a3:z ≤ −1
    then have a4:z < 0 by auto
    from a4 have h2:−z > 0
      using leD leI by auto
    from a3 have h5:−z ≥ 1
      by (simp add: leD leI)
    assume a5:x ≤ z
    then have h6:−x ≥ −z by auto
    have h3:−x ∗ −z = x ∗ z by auto
    show x ∗ y ≤ z
      using a1 a2 a3 a4 h1 h2 h3 h6 h5 a5 dual-order.trans less-eq-real-def
      by (metis mult-less-cancel-left1 not-le)
  qed
 have prereqs:r2 ≤ − 1 1 ≤ (real-of-int (sint w1))
            (− 2147483647) ≤ − (1::real ) r2 ≤ (− 2147483647)
  using rw1 rw2 HiNeg lower2 by (auto simp add: word-sless-def word-sle-def)
 have r1 ∗ r2 ≤ − 2147483647
  using upper lower1 lower2 rw1 rw2
  apply (auto simp add: word-sless-def word-sle-def)
  using mylem[of r2 (real-of-int (sint w1)) (− 2147483647)] prereqs
  by (auto simp add: mult.commute)
 then show ?thesis
  using HiNeg by(auto simp add: repe.simps)
next
 case AllFinite
 let ?prod = (((scast w1)::64 Word.word) ∗ ((scast w2)::64 Word.word))
 consider
    (ProdNeg) ?prod <=s ((scast NEG-INF)::64 Word.word)
  | (ProdPos) (((scast POS-INF)::64 Word.word) <=s ?prod)
  | (ProdFin) ¬(?prod <=s ((scast NEG-INF)::64 Word.word))
          ∧ ¬((scast POS-INF)::64 Word.word) <=s ?prod
  by (auto)
 then show ?thesis
 proof (cases)
  case ProdNeg
  have bigLeq:(4611686018427387904::real) ≤ 9223372036854775807 by auto
 have set-cast:⋀x::int. (x ∈ {−(2^31)..2^31}) = ( (real-of-int x) ∈ {−(2^31)..2^31})
    by auto
  have eq3:sint(((scast w1)::64 Word.word) ∗ ((scast w2)::64 Word.word)) =
   sint ((scast w1)::64 Word.word) ∗ sint ((scast w2)::64 Word.word)
   apply(rule Word-Lemmas.signed-arith-sint(4))
   using rangew1 rangew2 w1-cast w2-cast
   using Word.word-size[of ((scast w1)::64 Word.word)]
   using Word.word-size[of ((scast w2)::64 Word.word)]
   using times-upcast-upper[of sint w1 sint w2]
   using times-upcast-lower[of sint w1 sint w2]
   by auto
  assume ?prod <=s ((scast NEG-INF)::64 Word.word)
  then have sint-leq:sint ?prod ≤ sint ((scast NEG-INF)::64 Word.word)
```

```
    using word-sle-def by blast
   have neqs:w1 ≠ POS-INF  w1 ≠ NEG-INF w2 ≠ POS-INF w2 ≠ NEG-INF
    using AllFinite word-sless-def signed.not-less-iff-gr-or-eq by force+
  from eq1 have rw1:r1 = (real-of-int (sint w1)) using neqs by (auto simp add:
repe.simps)
  from eq2 have rw2:r2 = (real-of-int (sint w2)) using neqs by (auto simp add:
repe.simps)
   show ?thesis
    using AllFinite ProdNeg  w1-cast w2-cast rw1 rw2 sint-leq
    apply (auto simp add: repe.simps eq3)
    apply (subst (asm) of-int-le-iff [symmetric, where ?'a = real])
    apply simp
    done
 next
   case ProdPos
   have bigLeq:(4611686018427387904::real) ≤ 9223372036854775807 by auto
  have set-cast:⋀x::int. (x ∈ {-(2^31)..2^31}) = ( (real-of-int x) ∈ {-(2^31)..2^31})
    by auto
   have eq3:sint(((scast w1)::64 Word.word) * ((scast w2)::64 Word.word)) =
    sint ((scast w1)::64 Word.word) * sint ((scast w2)::64 Word.word)
    apply(rule Word-Lemmas.signed-arith-sint(4))
    using rangew1 rangew2 POS-cast POS-sint w1-cast w2-cast
    using Word.word-size[of ((scast w1)::64 Word.word)]
    using Word.word-size[of ((scast w2)::64 Word.word)]
    using times-upcast-upper[of sint w1 sint w2]
    using times-upcast-lower[of sint w1 sint w2]
    by auto
   assume cast:((scast POS-INF)::64 Word.word) <=s ?prod
   then have sint-leq:sint ((scast POS-INF)::64 Word.word) ≤ sint ?prod
    using word-sle-def by blast
   have neqs:w1 ≠ POS-INF  w1 ≠ NEG-INF w2 ≠ POS-INF w2 ≠ NEG-INF
    using AllFinite word-sless-def signed.not-less-iff-gr-or-eq by force+
   from eq1 have rw1:r1 =  (real-of-int (sint w1))
    using repe.simps AllFinite neqs by auto
   from eq2 have rw2:r2 = (real-of-int (sint w2))
    using repe.simps AllFinite neqs by auto
   have prodHi:r1 * r2 ≥ PosInf
    using w1-cast w2-cast rw1 rw2 sint-leq apply (auto simp add: eq3)
    apply (subst (asm) of-int-le-iff [symmetric, where ?'a = real])
    apply simp
    done
   have infs:SCAST(32 → 64) NEG-INF <s SCAST(32 → 64) POS-INF
    by (auto)
  have casted:SCAST(32 → 64) POS-INF <=s SCAST(32 → 64) w1 * SCAST(32
→ 64) w2
    using cast by auto
   have almostContra:SCAST(32 → 64) NEG-INF <s SCAST(32 → 64) w1 *
SCAST(32 → 64) w2
    using infs cast  signed.order.strict-trans2 by blast
```

```
  have contra:¬(SCAST(32 → 64) w1 * SCAST(32 → 64) w2 <=s SCAST(32
→ 64) NEG-INF)
    using eq3 almostContra by auto
   have wtimesCase:wtimes w1 w2 = POS-INF
    using neqs ProdPos almostContra wtimes.simps AllFinite ProdPos
    by (auto simp add: repe.simps Let-def)
   show ?thesis
    using prodHi
    apply(simp only: repe.simps)
    apply(rule disjI1)
    apply(rule exI[where x= r1*r2])
    apply(rule conjI)
     apply(rule wtimesCase)
    using prodHi by auto
 next
   case ProdFin
   have bigLeq:(4611686018427387904::real) ≤ 9223372036854775807 by auto
  have set-cast:⋀x::int. (x ∈ {−(2^31)..2^31}) = ( (real-of-int x) ∈ {−(2^31)..2^31})
    by auto
   have eq3:sint(((scast w1)::64 Word.word) * ((scast w2)::64 Word.word)) =
    sint ((scast w1)::64 Word.word) * sint ((scast w2)::64 Word.word)
    apply(rule Word-Lemmas.signed-arith-sint(4))
    using rangew1 rangew2 POS-cast POS-sint w1-cast w2-cast
    using Word.word-size[of ((scast w1)::64 Word.word)]
    using Word.word-size[of ((scast w2)::64 Word.word)]
    using times-upcast-upper[of sint w1 sint w2]
    using times-upcast-lower[of sint w1 sint w2]
    by auto
   from ProdFin have a1:¬(?prod <=s ((scast NEG-INF)::64 Word.word))
    by auto
   then have sintGe:sint (?prod) > sint (((scast NEG-INF)::64 Word.word))
    using word-sle-def dual-order.order-iff-strict signed.linear
    by fastforce
   from ProdFin have a2:¬((scast POS-INF)::64 Word.word) <=s ?prod
    by auto
   then have sintLe:sint (((scast POS-INF)::64 Word.word)) > sint (?prod)
    using word-sle-def dual-order.order-iff-strict signed.linear
    by fastforce
   have neqs:w1 ≠ POS-INF  w1 ≠ NEG-INF w2 ≠ POS-INF w2 ≠ NEG-INF
    using AllFinite word-sless-def signed.not-less-iff-gr-or-eq by force+
  from eq1 have rw1:r1 =  (real-of-int (sint w1)) using neqs by(auto simp add:
repe.simps)
  from eq2 have rw2:r2 =  (real-of-int (sint w2)) using neqs by(auto simp add:
repe.simps)
   from rw1 rw2 have r1 * r2 =  (real-of-int ((sint w1) * (sint w2)))
    by simp
   have rightSize:sint (((scast w1)::64 Word.word) * ((scast w2)::64 Word.word))
    ∈ sints (len-of TYPE(32))
    using sintLe sintGe sints32 by (simp)
```

```
    have downcast:sint ((scast (((scast w1)::64 Word.word) * ((scast w2)::64
Word.word)))::word)
              = sint (((scast w1)::64 Word.word) * ((scast w2)::64 Word.word))
    using scast-down-range[OF rightSize]
    by auto
  then have res-eq:r1 * r2
  = real-of-int(sint((scast (((scast w1)::64 Word.word)*((scast w2)::64 Word.word)))::word))
    using rw1 rw2 eq3 POS-cast POS-sint w1-cast w2-cast downcast
      ‹r1 * r2 =  (real-of-int (sint w1 * sint w2))›
    by (auto)
 have res-up:sint (scast (((scast w1)::64 Word.word) * ((scast w2)::64 Word.word))::word)

           < sint POS-INF
    using rw1 rw2 eq3 POS-cast POS-sint w1-cast w2-cast downcast
      ‹r1 * r2 = (real-of-int (sint w1 * sint w2))›
      ‹sint (scast w1 * scast w2) < sint (scast POS-INF)›
       of-int-eq-iff res-eq
    by presburger
  have res-lo:sint NEG-INF
        < sint (scast (((scast w1)::64 Word.word) * ((scast w2)::64 Word.word))::word)
    using rw1 rw2 eq3 POS-cast POS-sint w1-cast w2-cast NEG-cast downcast
      ‹r1 * r2 =  (real-of-int (sint w1 * sint w2))›
      ‹sint (scast NEG-INF) < sint (scast w1 * scast w2)›
      of-int-eq-iff res-eq
    by presburger
  have scast ?prod ≡E (r1 * r2)
    using res-eq res-up res-lo
    apply (auto simp add: rep-simps)
    using repeInt-simps by auto
  then show ?thesis
    using AllFinite ProdFin by(auto)
 qed
qed
qed
```

3.6 Multiplication upper bound

Upper bound of multiplication from upper and lower bounds

```
fun tu :: word ⇒ word ⇒ word ⇒ word ⇒ word
where tu w1l w1u w2l w2u =
 wmax (wmax (wtimes w1l w2l) (wtimes w1u w2l))
      (wmax (wtimes w1l w2u) (wtimes w1u w2u))

lemma tu-lemma:
  assumes u1:u1 ≡U (r1::real)
  assumes u2:u2 ≡U (r2::real)
  assumes l1:l1 ≡L (r1::real)
  assumes l2:l2 ≡L (r2::real)
  shows tu l1 u1 l2 u2 ≡U (r1 * r2)
```

proof −
obtain *rl1 rl2 ru1 ru2* :: *real*
where *gru1*:*ru1* ≥ *r1* **and** *gru2*:*ru2* ≥ *r2* **and** *grl1*:*rl1* ≤ *r1* **and** *grl2*:*rl2* ≤ *r2*

and *eru1*:u_1 ≡$_E$ *ru1* **and** *eru2*:u_2 ≡$_E$ *ru2* **and** *erl1*:l_1 ≡$_E$ *rl1* **and** *erl2*:l_2 ≡$_E$ *rl2*
using *u1 u2 l1 l2* **unfolding** *repU-def repL-def* **by** *auto*
have *timesuu*:*wtimes* u_1 u_2 ≡$_E$ *ru1* ∗ *ru2*
using *wtimes-exact*[*OF eru1 eru2*] **by** *auto*
have *timesul*:*wtimes* u_1 l_2 ≡$_E$ *ru1* ∗ *rl2*
using *wtimes-exact*[*OF eru1 erl2*] **by** *auto*
have *timeslu*:*wtimes* l_1 u_2 ≡$_E$ *rl1* ∗ *ru2*
using *wtimes-exact*[*OF erl1 eru2*] **by** *auto*
have *timesll*:*wtimes* l_1 l_2 ≡$_E$ *rl1* ∗ *rl2*
using *wtimes-exact*[*OF erl1 erl2*] **by** *auto*
have *maxt12*:*wmax* (*wtimes* l_1 l_2) (*wtimes* u_1 l_2) ≡$_E$ *max* (*rl1* ∗ *rl2*) (*ru1* ∗ *rl2*)
by (*rule wmax-lemma*[*OF timesll timesul*])
have *maxt34*:*wmax* (*wtimes* l_1 u_2) (*wtimes* u_1 u_2) ≡$_E$ *max* (*rl1* ∗ *ru2*) (*ru1* ∗ *ru2*)
by (*rule wmax-lemma*[*OF timeslu timesuu*])
have *bigMax*:*wmax* (*wmax* (*wtimes* l_1 l_2) (*wtimes* u_1 l_2)) (*wmax* (*wtimes* l_1 u_2) (*wtimes* u_1 u_2))
≡$_E$ *max* (*max* (*rl1* ∗ *rl2*) (*ru1* ∗ *rl2*)) (*max* (*rl1* ∗ *ru2*) (*ru1* ∗ *ru2*))
by (*rule wmax-lemma*[*OF maxt12 maxt34*])
obtain *maxt12val* :: *real*
where *maxU12*:*wmax* (*wtimes* l_1 l_2) (*wtimes* u_1 l_2) ≡$_U$ *max* (*rl1* ∗ *rl2*) (*ru1* ∗ *rl2*)
using *maxt12* **unfolding** *repU-def* **by** *blast*
obtain *maxt34val* :: *real*
where *maxU34*:*wmax* (*wtimes* l_1 u_2) (*wtimes* u_1 u_2) ≡$_U$ *max* (*rl1* ∗ *ru2*) (*ru1* ∗ *ru2*)
using *maxt34* **unfolding** *repU-def* **by** *blast*
obtain *bigMaxU*:*wmax* (*wmax* (*wtimes* l_1 l_2) (*wtimes* u_1 l_2)) (*wmax* (*wtimes* l_1 u_2) (*wtimes* u_1 u_2))
≡$_U$ *max* (*max* (*rl1* ∗ *rl2*) (*ru1* ∗ *rl2*)) (*max* (*rl1* ∗ *ru2*) (*ru1* ∗ *ru2*))
using *bigMax* **unfolding** *repU-def* **by** *blast*
have *ivl1*:*rl1* ≤ *ru1* **using** *grl1 gru1* **by** *auto*
have *ivl2*:*rl2* ≤ *ru2* **using** *grl2 gru2* **by** *auto*
let *?thesis* = *tu* l_1 u_1 l_2 u_2 ≡$_U$ *r1* ∗ *r2*
show *?thesis*
using *ivl1 ivl2*
proof(*cases rule*: *case-ivl-zero*)
case *ZeroZero*
assume *rl1* ≤ *0* ∧ *0* ≤ *ru1* ∧ *rl2* ≤ *0* ∧ *0* ≤ *ru2*
then have *geq1*:*ru1* ≥ *0* **and** *geq2*:*ru2* ≥ *0* **by** *auto*
consider *r1* ≥ *0* ∧ *r2* ≥ *0* | *r1* ≥ *0* ∧ *r2* ≤ *0* | *r1* ≤ *0* ∧ *r2* ≥ *0* | *r1* ≤ *0* ∧ *r2* ≤ *0*
using *le-cases* **by** *auto*
then show *tu* l_1 u_1 l_2 u_2 ≡$_U$ *r1* ∗ *r2*

```
  proof (cases)
    case 1
    have g1:ru1 * ru2 ≥ ru1 * r2
      using 1 geq1 geq2 grl2 gru2
      by (simp add: mult-left-mono)
    have g2:ru1 * r2 ≥ r1 * r2
      using 1 geq1 geq2 grl1 grl2 gru1 gru2
      by (simp add: mult-right-mono)
    from g1 and g2
    have up:ru1 * ru2 ≥ r1 * r2
      by auto
    show ?thesis
      using up eru1 eru2 erl1 erl2 repU-def timesuu tu.simps
      max-repU2[OF maxU12] max-repU2[OF maxU34] max-repU2[OF bigMaxU]
      by (metis wmax.elims)
  next
    case 2
    have g1:ru1 * ru2 ≥ 0
      using 2 geq1 geq2 grl2 gru2 by (simp)
    have g2:0 ≥ r1 * r2
      using 2 by (simp add: mult-le-0-iff)
      from g1 and g2
    have up:ru1 * ru2 ≥ r1 * r2 by auto
    show ?thesis
     using up maxU12 maxU34 bigMaxU wmax.elims max-repU2 max-repU2[OF
maxU12]
          max-repU2[OF maxU34] max-repU2[OF bigMaxU] max.coboundedI1
max.commute maxt34
      by (metis repU-def tu.simps)
  next
    case 3
    have g1:ru1 * ru2 ≥ 0
      using 3 geq1 geq2 by simp
    have g2:0 ≥ r1 * r2
      using 3 by (simp add: mult-le-0-iff)
    from g1 and g2
    have up:ru1 * ru2 ≥ r1 * r2 by auto
    show ?thesis
     using up maxU12 maxU34 bigMaxU wmax.elims max-repU2 max-repU2[OF
maxU12]
          max-repU2[OF maxU34] max-repU2[OF bigMaxU] repU-def tu.simps
timesuu
      by (metis max.coboundedI1 max.commute maxt34)
  next
    case 4
    have g1:rl1 * rl2 ≥ rl1 * r2
      using 4 geq1 geq2 grl1 grl2 gru1 gru2
      using ‹rl1 ≤ 0 ∧ 0 ≤ ru1 ∧ rl2 ≤ 0 ∧ 0 ≤ ru2› less-eq-real-def
        by (metis mult-left-mono-neg)
```

```
    have g2:rl1 * r2 ≥ r1 * r2
      using 4 geq1 geq2 grl1 grl2 gru1 gru2 ‹rl1 ≤ 0 ∧ 0 ≤ ru1 ∧ rl2 ≤ 0 ∧ 0
≤ ru2›
        by (metis mult-left-mono-neg mult.commute)
    from g1 and g2
    have up:rl1 * rl2 ≥ r1 * r2
      by auto
    show ?thesis
     using up maxU12 maxU34 bigMaxU wmax.elims max-repU2 max-repU2[OF
maxU12]
       max-repU2[OF maxU34] max-repU2[OF bigMaxU] max.commute maxt34
      by (metis max-repU1 repU-def timesll tu.simps)
    qed
  next
   case ZeroPos
   assume bounds:rl1 ≤ 0 ∧ 0 ≤ ru1 ∧ 0 ≤ rl2
   have r2:r2 ≥ 0 using bounds dual-order.trans grl2 by blast
   consider r1 ≥ 0 | r1 ≤ 0 using le-cases by (auto)
   then show ?thesis
   proof (cases)
    case 1
    assume r1:r1 ≥ 0
    have g1:ru1 * ru2 ≥ ru1 * r2
      using r1 r2 bounds grl1 grl2 gru1 gru2
      using mult-left-mono by blast
    have g2:ru1 * r2 ≥ r1 * r2
      using r1 r2 bounds grl1 grl2 gru1 gru2
      using mult-right-mono by blast
    from g1 and g2
    have up:ru1 * ru2 ≥ r1 * r2
      by auto
    show ?thesis
     using up maxU12 maxU34 bigMaxU wmax.elims max-repU2 max-repU2[OF
maxU12]
          max-repU2[OF maxU34] max-repU2[OF bigMaxU] max.coboundedI1
max.commute maxt34
      by (metis repU-def tu.simps)
   next
    case 2
    assume r1:r1 ≤ 0
    have g1:ru1 * ru2 ≥ 0
      using r1 r2 bounds grl1 grl2 gru1 gru2
      using mult-left-mono
      by (simp add: mult-less-0-iff less-le-trans not-less)
    have g2:0 ≥ r1 * r2
      using r1 r2 bounds grl1 grl2 gru1 gru2
      using mult-right-mono
      by (simp add: mult-le-0-iff)
    from g1 and g2
```

```
    have up:ru1 * ru2 ≥ r1 * r2
      by auto
    show ?thesis
     using up maxU12 maxU34 bigMaxU wmax.elims max-repU2 max-repU2[OF
maxU12]
           max-repU2[OF maxU34] max-repU2[OF bigMaxU] max.coboundedI1
max.commute maxt34
      by (metis repU-def tu.simps)
    qed
  next
    case ZeroNeg
    assume bounds:rl1 ≤ 0 ∧ 0 ≤ ru1 ∧ ru2 ≤ 0
    have r2:r2 ≤ 0 using bounds dual-order.trans gru2 by blast
    have case1:r1 ≥ 0 ⟹ ?thesis
    proof −
      assume r1:r1 ≥ 0
      have g1:rl1 * rl2 ≥ 0
        using r1 r2 bounds grl1 grl2 gru1 gru2 mult-less-0-iff less-le-trans not-less
        by metis
      have g2:0 ≥ r1 * r2
        using r1 r2 bounds grl1 grl2 gru1 gru2
        using mult-right-mono
        by (simp add: mult-le-0-iff)
      from g1 and g2
      have up:rl1 * rl2 ≥ r1 * r2
        by auto
      show ?thesis
       using up maxU12 maxU34 bigMaxU wmax.elims max-repU2 max-repU2[OF
maxU12]
         max-repU2[OF maxU34] max-repU2[OF bigMaxU] max.commute maxt34
        by (metis max-repU2 max-repU1 repU-def timesll tu.simps)
    qed
    have case2:r1 ≤ 0 ⟹ ?thesis
    proof −
      assume r1:r1 ≤ 0
      have g1:rl1 * rl2 ≥ rl1 * r2
        using r1 r2 bounds  grl1 grl2 gru1 gru2
        by (metis mult-left-mono-neg)
      have g2:rl1 * r2 ≥ r1 * r2
        using r1 r2 bounds grl1 grl2 gru1 gru2 mult.commute
        by (metis mult-left-mono-neg)
      from g1 and g2
      have up:rl1 * rl2 ≥ r1 * r2
        by auto
      show ?thesis
       using up maxU12 maxU34 bigMaxU wmax.elims max-repU2 max-repU2[OF
maxU12]
         max-repU2[OF maxU34] max-repU2[OF bigMaxU] max.commute maxt34
        by (metis max-repU1 repU-def timesll tu.simps)
```

```
    qed
    show tu l1 u1 l2 u2 ≡U r1 * r2
      using case1 case2 le-cases by blast
  next
    case PosZero
    assume bounds:0 ≤ rl1 ∧ rl2 ≤ 0 ∧ 0 ≤ ru2
    have r1:r1 ≥ 0 using bounds dual-order.trans grl1 by blast
    consider r2 ≥ 0 | r2 ≤ 0 using le-cases by auto
    then show ?thesis
    proof (cases)
      case 1
      have g1:ru1 * ru2 ≥ ru1 * r2
        using 1 bounds grl1 grl2 gru1 gru2
        using mult-left-mono
        using leD leI less-le-trans by metis
      have g2:ru1 * r2 ≥ r1 * r2
        using 1 bounds grl1 grl2 gru1 gru2
        using mult-right-mono by blast
      from g1 and g2
      have up:ru1 * ru2 ≥ r1 * r2
        by auto
      show ?thesis
       using up maxU12 maxU34 bigMaxU wmax.elims max-repU2 max-repU2[OF
maxU12]
            max-repU2[OF maxU34] max-repU2[OF bigMaxU] max.coboundedI1
max.commute maxt34
       by (metis repU-def tu.simps)
    next
      case 2
       have g1:ru1 * ru2 ≥ 0
         using r1 bounds grl2 gru2 gru1 leD leI less-le-trans by auto
       have g2:0 ≥ r1 * r2
         using r1 2
         by (simp add: mult-le-0-iff)
       from g1 and g2
       have up:ru1 * ru2 ≥ r1 * r2
         by auto
       show ?thesis
        using up maxU12 maxU34 bigMaxU wmax.elims max-repU2 max-repU2[OF
maxU12]
             max-repU2[OF maxU34] max-repU2[OF bigMaxU] max.coboundedI1
max.commute maxt34
         by (metis repU-def tu.simps)
      qed
  next
    case NegZero
    assume bounds:ru1 ≤ 0 ∧ rl2 ≤ 0 ∧ 0 ≤ ru2
    have r1:r1 ≤ 0  using bounds dual-order.trans gru1 by blast
    consider r2 ≥ 0 | r2 ≤ 0 using le-cases by auto
```

```
    then show ?thesis
    proof (cases)
      case 1
      have g1:ru1 * rl2 ≥ 0
        using r1 1 bounds grl1 grl2 gru1 gru2 mult-less-0-iff not-less
        by metis
      have g2:0 ≥ r1 * r2
        using r1 1 bounds grl1 grl2 gru1 gru2
        by (simp add: mult-le-0-iff)
      from g1 and g2
      have up:ru1 * rl2 ≥ r1 * r2
        by auto
      show ?thesis
       using up maxU12 maxU34 bigMaxU wmax.elims max-repU2 max-repU2[OF
maxU12]
         max-repU2[OF maxU34] max-repU2[OF bigMaxU] max.commute maxt34
        by (metis max-repU1 repU-def timesul tu.simps)
    next
      case 2
      have lower:rl1 ≤ 0 using bounds dual-order.trans grl1 r1 by blast
      have g1:rl1 * rl2 ≥ rl1 * r2
        using r1 2 bounds grl1 grl2 gru1 gru2 less-eq(1) less-le-trans not-less
          mult-le-cancel-left
        by metis
      have g2:rl1 * r2 ≥ r1 * r2
      using r1 2 bounds grl1 grl2 gru1 gru2 mult.commute not-le lower mult-le-cancel-left
        by metis
      from g1 and g2
      have up:rl1 * rl2 ≥ r1 * r2
        by auto
      show ?thesis
       using up maxU12 maxU34 bigMaxU wmax.elims max-repU2 max-repU2[OF
maxU12]
         max-repU2[OF maxU34] max-repU2[OF bigMaxU] max.commute maxt34
        by (metis max-repU1 repU-def timesll tu.simps)
    qed
  next
    case NegNeg
    assume bounds:ru1 ≤ 0 ∧ ru2 ≤ 0
    have r1:r1 ≤ 0 using bounds dual-order.trans gru1 by blast
    have r2:r2 ≤ 0 using bounds dual-order.trans gru2 by blast
    have lower1:rl1 ≤ 0 using bounds dual-order.trans grl1 r1 by blast
    have lower2:rl2 ≤ 0 using bounds dual-order.trans grl2 r2 by blast
    have g1:rl1 * rl2 ≥ rl1 * r2
     using r1 r2 bounds  grl1 grl2 gru1 gru2 less-eq(1)  mult-le-cancel-left less-le-trans
not-less
      by metis
    have g2:rl1 * r2 ≥ r1 * r2
       using r1 r2 bounds grl1 grl2 gru1 gru2 mult.commute not-le lower1 lower2
```

mult-le-cancel-left
by *metis*
from *g1* **and** *g2*
have *up*:*rl1* $*$ *rl2* $\geq$ *r1* $*$ *r2*
by *auto*
show *?thesis*
using *up maxU12 maxU34 bigMaxU wmax.elims max-repU2 max-repU2*[*OF maxU12*]
max-repU2[*OF maxU34*] *max-repU2*[*OF bigMaxU*] *max.commute maxt34*
by (*metis max-repU1 repU-def timesll tu.simps*)
next
case *NegPos*
assume *bounds*:*ru1* $\leq$ *0* $\wedge$ *0* $\leq$ *rl2*
have *r1*:*r1* $\leq$ *0* **using** *bounds dual-order.trans gru1* **by** *blast*
have *r2*:*r2* $\geq$ *0* **using** *bounds dual-order.trans grl2* **by** *blast*
have *lower1*:*rl1* $\leq$ *0* **using** *bounds dual-order.trans grl1 r1* **by** *blast*
have *lower2*:*rl2* $\geq$ *0* **using** *bounds* **by** *auto*
have *upper1*:*ru1* $\leq$ *0* **using** *bounds* **by** *auto*
have *upper2*:*ru2* $\geq$ *0* **using** *bounds dual-order.trans gru2 r2* **by** *blast*
have *g1*:*ru1* $*$ *rl2* $\geq$ *ru1* $*$ *r2*
using *r1 r2 bounds grl1 grl2 gru1 gru2 not-less upper1 lower2 mult-le-cancel-left*
by *metis*
have *g2*:*ru1* $*$ *r2* $\geq$ *r1* $*$ *r2*
using *r1 upper1 r2 mult-right-mono gru1* **by** *metis*
from *g1* **and** *g2*
have *up*:*ru1* $*$ *rl2* $\geq$ *r1* $*$ *r2*
by *auto*
show *?thesis*
using *up maxU12 maxU34 bigMaxU wmax.elims maxt34*
max-repU2[*OF maxU12*] *max-repU2*[*OF maxU34*] *max-repU2*[*OF bigMaxU*]
by (*metis max-repU1 repU-def timesul tu.simps*)
next
case *PosNeg*
assume *bounds*:*0* $\leq$ *rl1* $\wedge$ *ru2* $\leq$ *0*
have *r1*:*r1* $\geq$ *0* **using** *bounds dual-order.trans grl1* **by** *blast*
have *r2*:*r2* $\leq$ *0* **using** *bounds dual-order.trans gru2* **by** *blast*
have *lower1*:*rl1* $\geq$ *0* **using** *bounds* **by** *auto*
have *lower2*:*rl2* $\leq$ *0* **using** *dual-order.trans grl2 r2* **by** *blast*
have *upper1*:*ru1* $\geq$ *0* **using** *dual-order.trans gru1 u1 r1* **by** *blast*
have *upper2*:*ru2* $\leq$ *0* **using** *bounds* **by** *auto*
have *g1*:*rl1* $*$ *ru2* $\geq$ *rl1* $*$ *r2*
using *r1 r2 bounds grl1 grl2 gru1 gru2 not-less upper2 lower1 mult-le-cancel-left*

by *metis*
have *g2*:*rl1* $*$ *r2* $\geq$ *r1* $*$ *r2*
using *r1 lower1 r2 not-less gru2 gru1 grl1 grl2*
by (*metis mult-le-cancel-left mult.commute*)
from *g1* **and** *g2*
have *up*:*rl1* $*$ *ru2* $\geq$ *r1* $*$ *r2*

by *auto*
show *tu* l_1 u_1 l_2 u_2 $\equiv_U$ *r1* $*$ *r2*
using *up maxU12 maxU34 bigMaxU wmax.elims max.coboundedI1 max.commute maxt34*
max-repU2[*OF maxU12*] *max-repU2*[*OF maxU34*] *max-repU2*[*OF bigMaxU*]
by (*metis repU-def tu.simps*)
next
case *PosPos*
assume *bounds*:*0* $\leq$ *rl1* $\wedge$ *0* $\leq$ *rl2*
have *r1*:*r1* $\geq$ *0* **using** *bounds dual-order.trans grl1* **by** *blast*
have *r2*:*r2* $\geq$ *0* **using** *bounds dual-order.trans grl2* **by** *blast*
have *lower1*:*rl1* $\geq$ *0* **using** *bounds* **by** *auto*
have *lower2*:*rl2* $\geq$ *0* **using** *bounds* **by** *auto*
have *upper1*:*ru1* $\geq$ *0* **using** *dual-order.trans gru1 u1 r1* **by** *blast*
have *upper2*:*ru2* $\geq$ *0* **using** *dual-order.trans gru2 u2 r2 bounds* **by** *blast*
have *g1*:*ru1* $*$ *ru2* $\geq$ *ru1* $*$ *r2*
using *r1 r2 bounds grl1 grl2 gru1 gru2 mult-left-mono leD leI less-le-trans* **by** *metis*
have *g2*:*ru1* $*$ *r2* $\geq$ *r1* $*$ *r2*
using *r1 r2 bounds grl1 grl2 gru1 gru2 mult-right-mono* **by** *metis*
from *g1* **and** *g2*
have *up*:*ru1* $*$ *ru2* $\geq$ *r1* $*$ *r2*
by *auto*
show *?thesis*
using *up maxU12 maxU34 bigMaxU wmax.elims max.coboundedI1 max.commute maxt34*
max-repU2[*OF bigMaxU*] *max-repU2*[*OF maxU12*] *max-repU2*[*OF maxU34*]
by (*metis repU-def tu.simps*)
qed
qed

3.7 Minimum function

Minimum of 2s-complement words

fun *wmin* :: *word* $\Rightarrow$ *word* $\Rightarrow$ *word*
where *wmin w1 w2* =
(*if w1* $<_s$ *w2 then w1 else w2*)

Correctness of wmin

lemma *wmin-lemma*:
assumes *eq1*:*w1* $\equiv_E$ (*r1*::*real*)
assumes *eq2*:*w2* $\equiv_E$ (*r2*::*real*)
shows *wmin w1 w2* $\equiv_E$ (*min r1 r2*)
proof(*cases rule*: *case-inf2*[**where** *?w1.0*=*w1*, **where** *?w2.0*=*w2*])
case *PosPos*
assume *p1*:*w1* = *POS-INF*
and *p2*:*w2* = *POS-INF*
then have *bound1*:(*real-of-int* (*sint POS-INF*)) $\leq$ *r1*

```
      and bound2:(real-of-int (sint POS-INF)) ≤ r2
    using eq1 eq2 by (auto simp add: rep-simps repe.simps)
  have eqInf:wmin w1 w2 = POS-INF
    using p1 p2 unfolding wmin.simps by auto
  have pos-eq:POS-INF ≡E min r1 r2
    apply(rule repPOS-INF)
    using bound1 bound2 unfolding eq1 eq2 by auto
  show ?thesis
    using pos-eq eqInf by auto
next
  case PosNeg
  assume p1:w1 = POS-INF
  assume n2:w2 = NEG-INF
  obtain r ra :: real
  where bound1: (real-of-int (sint POS-INF)) ≤ r
    and bound2:ra ≤  (real-of-int (sint NEG-INF))
    and eq1:r1 = r
    and eq2:r2 = ra
    using p1 n2 eq1 eq2 by(auto simp add: rep-simps repe.simps)
  have eqNeg:wmin w1 w2 = NEG-INF
    unfolding eq1 eq2 wmin.simps p1 n2 word-sless-def word-sle-def
    by(auto)
  have neg-eq:NEG-INF ≡E min r1 r2
    apply(rule repNEG-INF)
    using bound1 bound2 eq1 eq2 by auto
  show ?thesis
    using eqNeg neg-eq by auto
next
  case PosNum
  assume p1:w1 = POS-INF
  assume np2:w2 ≠ POS-INF
  assume nn2:w2 ≠ NEG-INF
  have eq2:r2 = (real-of-int (sint w2))
    and bound1:(real-of-int (sint POS-INF)) ≤ r1
    and bound2a:(real-of-int (sint NEG-INF)) < (real-of-int (sint w2))
    and bound2b:(real-of-int (sint w2)) < (real-of-int (sint POS-INF))
    using p1 np2 nn2 eq1 eq2 by(auto simp add: rep-simps repe.simps)
  have eqNeg:min r1 r2 = sint w2
    using p1
    by (metis bound1 bound2b dual-order.trans eq2 min-def not-less)
  have neg-eq:wmin w1 w2 ≡E  (real-of-int (sint (wmin w1 w2)))
    apply (rule repINT)
    using bound1 bound2a bound2b bound2b p1 unfolding eq1 eq2
    by (auto simp add: word-sless-alt)
  show ?thesis
    using eqNeg neg-eq
   by (metis bound2b less-eq-real-def not-less of-int-less-iff p1 wmin.simps word-sless-alt)
next
  case NegPos
```

```
  assume n1:w1 = NEG-INF
  assume p2:w2 = POS-INF
  have  bound1:r1 ≤ (real-of-int (sint NEG-INF))
    and bound2:(real-of-int (sint POS-INF)) ≤ r2
    using n1 p2 eq1 eq2 by(auto simp add: rep-simps repe.simps)
  have eqNeg:wmin w1 w2 = NEG-INF
    unfolding eq1 eq2 wmin.simps n1 p2 word-sless-def word-sle-def
    by(auto)
  have neg-eq:NEG-INF ≡E min r1 r2
    apply(rule repNEG-INF)
    using bound1 bound2 unfolding eq1 eq2 by auto
  show wmin w1 w2 ≡E min r1 r2
    using eqNeg neg-eq by auto
next
  case NegNeg
  assume n1:w1 = NEG-INF
  assume n2:w2 = NEG-INF
  have bound1:r1 ≤  (real-of-int (sint NEG-INF))
   and bound2:r2 ≤  (real-of-int (sint NEG-INF))
    using n1 n2 eq1 eq2 by(auto simp add: rep-simps repe.simps)
  have eqNeg:NEG-INF ≡E min r1 r2
    apply(rule repNEG-INF)
    using eq1 eq2 bound1 bound2 unfolding NEG-INF-def
    by (auto)
  have neg-eq:wmin w1 w2 = NEG-INF
    using n1 n2 unfolding NEG-INF-def wmin.simps by auto
  show wmin w1 w2 ≡E min r1 r2
    using eqNeg neg-eq by auto
next
  case NegNum
  assume n1:w1 = NEG-INF
    and  nn2:w2 ≠ NEG-INF
    and  np2:w2 ≠ POS-INF
  have eq2:r2 =  (real-of-int (sint w2))
    and bound2a:(real-of-int (sint w2)) <  (real-of-int (sint POS-INF))
    and bound2b:(real-of-int (sint NEG-INF)) <  (real-of-int (sint w2))
    and bound1:r1 ≤  (real-of-int (sint NEG-INF))
    using n1 nn2 np2 eq2 eq1 eq2 by (auto simp add: rep-simps repe.simps)
  have eqNeg:wmin w1 w2 = NEG-INF
    using n1 assms(2) bound2a eq2 n1 repeInt-simps
    by (auto simp add: word-sless-alt)
  have neg-eq:NEG-INF ≡E min r1 r2
    apply(rule repNEG-INF)
    using bound1 bound2a bound2b eq1 min-le-iff-disj by blast
  show wmin w1 w2 ≡E min r1 r2
    using eqNeg neg-eq by auto
next
  case NumPos
  assume p2:w2 = POS-INF
```

```
  and  nn1:w1 ≠ NEG-INF
  and  np1:w1 ≠ POS-INF
 have eq1:r1 =  (real-of-int (sint w1))
  and bound1a: (real-of-int (sint w1)) <  (real-of-int (sint POS-INF))
  and bound1b: (real-of-int (sint NEG-INF)) <  (real-of-int (sint w1))
  and bound2: (real-of-int (sint POS-INF)) ≤ r2
  using nn1 np1 p2 eq2 eq1 eq2 by (auto simp add: rep-simps repe.simps)
 have res1:wmin w1 w2 = w1
  using p2 eq1 eq2  assms(1) bound1b p2 repeInt-simps
  by (auto simp add: word-sless-alt)
 have res2:min r1 r2 =  (real-of-int (sint w1))
  using eq1 eq2 bound1a bound1b bound2
  by transfer (auto simp add: less-imp-le less-le-trans min-def)
 have res3:wmin w1 w2 ≡E  (real-of-int (sint (wmin w1 w2)))
  apply(rule repINT)
  using p2 bound1a res1 bound1a bound1b bound2
  by auto
 show wmin w1 w2 ≡E min r1 r2
  using res1 res2 res3 by auto
next
 case NumNeg
 assume nn1:w1 ≠ NEG-INF
 assume np1:w1 ≠ POS-INF
 assume n2:w2 = NEG-INF
 have eq1:r1 =  (real-of-int (sint w1))
  and bound1a: (real-of-int (sint w1)) <  (real-of-int (sint POS-INF))
  and bound1b: (real-of-int (sint NEG-INF)) <  (real-of-int (sint w1))
  and bound2:r2 ≤  (real-of-int (sint NEG-INF))
  using nn1 np1 n2 eq2 eq1 eq2 by (auto simp add: rep-simps repe.simps)
 have res1:wmin w1 w2 = NEG-INF
  using n2 bound1b
 by (metis min.absorb-iff2 min-def n2 not-less of-int-less-iff wmin.simps word-sless-alt)
 have res2:NEG-INF ≡E min r1 r2
  apply(rule repNEG-INF)
  using eq1 eq2 bound1a bound1b bound2 min-le-iff-disj by blast
 show wmin w1 w2 ≡E min r1 r2
  using res1 res2 by auto
next
 case NumNum
 assume np1:w1 ≠ POS-INF
 assume nn1:w1 ≠ NEG-INF
 assume np2:w2 ≠ POS-INF
 assume nn2:w2 ≠ NEG-INF
 have eq1:r1 =  (real-of-int (sint w1))
 and  eq2:r2 =  (real-of-int (sint w2))
 and  bound1a: (real-of-int (sint w1)) <  (real-of-int (sint POS-INF))
 and  bound1b: (real-of-int (sint NEG-INF)) <  (real-of-int (sint w1))
 and  bound2a: (real-of-int (sint w2)) <  (real-of-int (sint POS-INF))
 and  bound2b: (real-of-int (sint NEG-INF)) <  (real-of-int (sint w2))
```

using *nn1 np1 nn2 np2 eq2 eq1 eq2*
by (*auto simp add*: *rep-simps repe.simps*)
have *res1*:*min r1 r2* = (*real-of-int* (*sint* (*wmin w1 w2*)))
using *eq1 eq2 bound1a bound1b bound2a bound2b*
apply (*simp add*: *min-def word-sless-alt not-less*)
apply *transfer*
apply *simp*
done
have *res2*:*wmin w1 w2* $\equiv_E$ (*real-of-int* (*sint* (*wmin w1 w2*)))
apply (*rule repINT*)
using *bound1a bound1b bound2a bound2b*
by (*simp add*: ‹*min r1 r2* = (*real-of-int* (*sint* (*wmin w1 w2*)))› *eq2 min-less-iff-disj*)+
show *wmin w1 w2* $\equiv_E$ *min r1 r2*
using *res1 res2* **by** *auto*
qed

lemma *min-repU1*:
assumes *w1* $\equiv_L$ *x*
assumes *w2* $\equiv_L$ *y*
shows *wmin w1 w2* $\equiv_L$ *x*
using *wmin-lemma assms repL-def*
by (*meson min-le-iff-disj*)

lemma *min-repU2*:
assumes *w1* $\equiv_L$ *y*
assumes *w2* $\equiv_L$ *x*
shows *wmin w1 w2* $\equiv_L$ *x*
using *wmin-lemma assms repL-def*
by (*meson min-le-iff-disj*)

3.8 Multiplication lower bound

Multiplication lower bound

fun *tl* :: *word* ⇒ *word* ⇒ *word* ⇒ *word* ⇒ *word*
where *tl w1l w1u w2l w2u* =
wmin (*wmin* (*wtimes w1l w2l*) (*wtimes w1u w2l*))
(*wmin* (*wtimes w1l w2u*) (*wtimes w1u w2u*))

Correctness of multiplication lower bound

lemma *tl-lemma*:
assumes *u1*:u_1 $\equiv_U$ (*r1*::*real*)
assumes *u2*:u_2 $\equiv_U$ (*r2*::*real*)
assumes *l1*:l_1 $\equiv_L$ (*r1*::*real*)
assumes *l2*:l_2 $\equiv_L$ (*r2*::*real*)
shows *tl* l_1 u_1 l_2 u_2 $\equiv_L$ (*r1* ∗ *r2*)
proof −
obtain *rl1 rl2 ru1 ru2* :: *real*
where *gru1*:*ru1* ≥ *r1* **and** *gru2*:*ru2* ≥ *r2* **and** *grl1*:*rl1* ≤ *r1* **and** *grl2*:*rl2* ≤ *r2*

and *eru1*:$u_1 \equiv_E$ *ru1* **and** *eru2*:$u_2 \equiv_E$ *ru2* **and** *erl1*:$l_1 \equiv_E$ *rl1* **and** *erl2*:$l_2 \equiv_E$ *rl2*
using *u1 u2 l1 l2* **unfolding** *repU-def repL-def* **by** *auto*
have *timesuu*:*wtimes* u_1 $u_2 \equiv_E$ *ru1* $*$ *ru2*
using *wtimes-exact*[*OF eru1 eru2*] **by** *auto*
have *timesul*:*wtimes* u_1 $l_2 \equiv_E$ *ru1* $*$ *rl2*
using *wtimes-exact*[*OF eru1 erl2*] **by** *auto*
have *timeslu*:*wtimes* l_1 $u_2 \equiv_E$ *rl1* $*$ *ru2*
using *wtimes-exact*[*OF erl1 eru2*] **by** *auto*
have *timesll*:*wtimes* l_1 $l_2 \equiv_E$ *rl1* $*$ *rl2*
using *wtimes-exact*[*OF erl1 erl2*] **by** *auto*
have *maxt12*:*wmin* (*wtimes* l_1 l_2) (*wtimes* u_1 l_2) $\equiv_E$ *min* (*rl1* $*$ *rl2*) (*ru1* $*$ *rl2*)
by (*rule wmin-lemma*[*OF timesll timesul*])
have *maxt34*:*wmin* (*wtimes* l_1 u_2) (*wtimes* u_1 u_2) $\equiv_E$ *min* (*rl1* $*$ *ru2*) (*ru1* $*$ *ru2*)
by (*rule wmin-lemma*[*OF timeslu timesuu*])
have *bigMax*:*wmin* (*wmin* (*wtimes* l_1 l_2) (*wtimes* u_1 l_2)) (*wmin* (*wtimes* l_1 u_2) (*wtimes* u_1 u_2))
$\equiv_E$ *min* (*min*(*rl1* $*$ *rl2*) (*ru1* $*$ *rl2*)) (*min* (*rl1* $*$ *ru2*) (*ru1* $*$ *ru2*))
by (*rule wmin-lemma*[*OF maxt12 maxt34*])
obtain *maxt12val* :: *real*
where *maxU12*:*wmin* (*wtimes* l_1 l_2) (*wtimes* u_1 l_2) $\equiv_L$ *min* (*rl1* $*$ *rl2*) (*ru1* $*$ *rl2*)
using *maxt12* **unfolding** *repL-def* **by** *blast*
obtain *maxt34val* :: *real*
where *maxU34*:*wmin* (*wtimes* l_1 u_2) (*wtimes* u_1 u_2) $\equiv_L$ *min* (*rl1* $*$ *ru2*) (*ru1* $*$ *ru2*)
using *maxt34* **unfolding** *repL-def* **by** *blast*
obtain *bigMaxU*:*wmin* (*wmin* (*wtimes* l_1 l_2) (*wtimes* u_1 l_2)) (*wmin* (*wtimes* l_1 u_2) (*wtimes* u_1 u_2))
$\equiv_L$ *min* (*min* (*rl1* $*$ *rl2*) (*ru1* $*$ *rl2*)) (*min* (*rl1* $*$ *ru2*) (*ru1* $*$ *ru2*))
using *bigMax* **unfolding** *repL-def* **by** *blast*
have *ivl1*:*rl1* $\leq$ *ru1* **using** *grl1 gru1* **by** *auto*
have *ivl2*:*rl2* $\leq$ *ru2* **using** *grl2 gru2* **by** *auto*
let *?thesis* = *tl* l_1 u_1 l_2 $u_2 \equiv_L$ *r1* $*$ *r2*
show *?thesis*
using *ivl1 ivl2*
proof(*cases rule*: *case-ivl-zero*)
case *ZeroZero*
assume *rl1* $\leq$ *0* $\wedge$ *0* $\leq$ *ru1* $\wedge$ *rl2* $\leq$ *0* $\wedge$ *0* $\leq$ *ru2*
then have *geq1*:*ru1* $\geq$ *0* **and** *geq2*:*ru2* $\geq$ *0*
and *geq3*:*rl1* $\leq$ *0* **and** *geq4*:*rl2* $\leq$ *0* **by** *auto*
consider *r1* $\geq$ *0* $\wedge$ *r2* $\geq$ *0* | *r1* $\leq$ *0* $\wedge$ *r2* $\geq$ *0* | *r1* $\geq$ *0* $\wedge$ *r2* $\leq$ *0* | *r1* $\leq$ *0* $\wedge$ *r2* $\leq$ *0*
using *le-cases* **by** *auto*
then show *?thesis*
proof (*cases*)
case *1*
have *g1*:*rl1* $*$ *ru2* $\leq$ *0*

```
      using 1 geq1 geq2 geq3 geq4 grl2 gru2 mult-le-0-iff by blast
    have g2:0 ≤ r1 * r2
      using 1 geq1 geq2 grl1 grl2 gru1 gru2
      by (simp)
    from g1 and g2
    have up:rl1 * ru2 ≤ r1 * r2
      by auto
    show ?thesis
      using up eru1 eru2 erl1 erl2 min-repU1 min-repU2
        repL-def repU-def timeslu tl.simps wmin.elims
      by (metis bigMax min-le-iff-disj)
  next
    case 2
    have g1:rl1 * ru2 ≤ rl1 * r2
      using 2 geq1 geq2 grl2 gru2
      by (metis mult-le-cancel-left geq3 leD)
    have g2:rl1 * r2 ≤ r1 * r2
      using 2 geq1 geq2 grl2 gru2
      by (simp add: mult-right-mono grl1)
    from g1 and g2
    have up:rl1 * ru2 ≤ r1 * r2
      by auto
    show ?thesis
     by (metis up maxU12 min-repU2 repL-def tl.simps min.coboundedI1 maxt34)
  next
    case 3
    have g1:ru1 * rl2 ≤ ru1 * r2
      using 3 geq1 geq2 grl2 gru2
      by (simp add: mult-left-mono)
    have g2:ru1 * r2 ≤ r1 * r2
      using 3 geq1 geq2 grl1 grl2 gru1 gru2 mult-minus-right mult-right-mono
      by (simp add: mult-right-mono-neg)
    from g1 and g2
    have up:ru1 * rl2 ≤ r1 * r2 by auto
    show ?thesis
       using up maxU12 maxU34 bigMaxU wmin.elims min-repU2 min-repU1
maxt34 timesul
      by (metis repL-def tl.simps)
  next
    case 4
    have g1:ru1 * rl2 ≤ 0
     using 4 geq1 geq2 grl1 grl2 gru1 gru2 ‹rl1 ≤ 0 ∧ 0 ≤ ru1 ∧ rl2 ≤ 0 ∧ 0
≤ ru2›
       mult-less-0-iff less-eq-real-def not-less
      by auto
    have g2:0 ≤ r1 * r2
      using 4 geq1 geq2 grl1 grl2 gru1 gru2
      by (metis mult-less-0-iff not-less)
    from g1 and g2
```

```
      have up:ru1 * rl2 ≤ r1 * r2
        by auto
      show ?thesis
       by (metis up maxU12 maxU34 wmin.elims min-repU1 min-repU2 repL-def
timesul tl.simps)
    qed
  next
    case ZeroPos
    assume bounds:rl1 ≤ 0 ∧ 0 ≤ ru1 ∧ 0 ≤ rl2
    have r2:r2 ≥ 0 using bounds dual-order.trans grl2 by blast
    consider r1 ≥ 0 | r1 ≤ 0 using le-cases by auto
    then show ?thesis
    proof (cases)
      case 1
      have g1:rl1 * rl2 ≤ 0
        using 1 r2 bounds grl1 grl2 gru1 gru2
        by (simp add: mult-le-0-iff)
      have g2:0 ≤ r1 * r2
        using 1 r2 bounds grl1 grl2 gru1 gru2
        by (simp)
      from g1 and g2
      have up:rl1 * rl2 ≤ r1 * r2
        by auto
      show ?thesis
      by (metis repL-def timesll tl.simps up maxU12 maxU34 wmin.elims min-repU2
min-repU1)
    next
      case 2
      have bound:ru2 ≥ 0
        using 2 r2 bounds grl1 grl2 gru1 gru2 dual-order.trans by auto
      then have g1:rl1 * ru2 ≤ rl1 * r2
        using 2 r2 bounds grl1 grl2 gru1 gru2 mult-le-cancel-left
        by fastforce
      have g2:rl1 * r2 ≤ r1 * r2
        using 2 r2 bounds grl1 grl2 gru1 gru2 mult-le-0-iff mult-le-cancel-right by
fastforce
      from g1 and g2
      have up:rl1 * ru2 ≤ r1 * r2 by auto
      show ?thesis
          by (metis up maxU12 wmin.elims min-repU2 min.coboundedI1 maxt34
repL-def tl.simps)
      qed
  next
    case ZeroNeg
    assume bounds:rl1 ≤ 0 ∧ 0 ≤ ru1 ∧ ru2 ≤ 0
    have r2:r2 ≤ 0 using bounds dual-order.trans gru2 by blast
    consider (Pos) r1 ≥ 0 | (Neg) r1 ≤ 0 using le-cases by auto
    then show ?thesis
    proof (cases)
```

```
      case Pos
      have bound:rl2 ≤ 0
        using Pos r2 bounds grl1 grl2 gru1 gru2 dual-order.trans by auto
      then have g1:ru1 * rl2 ≤ ru1 * r2
        using Pos bounds grl1 grl2 gru1 gru2 mult-le-cancel-left
        by fastforce
      have p1:⋀a::real. (0 ≤ − a) = (a ≤ 0)
        by(auto)
      have p2:⋀a b::real.  (− a ≤ − b) = (b ≤ a) by auto
      have g2:ru1 * r2 ≤ r1 * r2
        using Pos r2 bounds grl1 grl2 gru1 gru2 p1 p2
        by (simp add: mult-right-mono-neg)
      from g1 and g2
      have up:ru1 * rl2 ≤ r1 * r2
        by auto
      show ?thesis
        by (metis up maxU12 maxU34 wmin.elims min-repU2 min-repU1 repL-def
timesul tl.simps)
    next
      case Neg
      have g1:ru1 * ru2 ≤ 0
        using Neg r2 bounds  grl1 grl2 gru1 gru2 mult-le-0-iff by blast
      have g2:0 ≤ r1 * r2
        using Neg r2 zero-le-mult-iff by blast
      from g1 and g2
      have up:ru1 * ru2 ≤ r1 * r2
        by auto
      show ?thesis
        using up maxU12 maxU34 bigMaxU wmin.elims min-repU2 min-repU1
        min.coboundedI1 min.commute maxt34
        by (metis repL-def tl.simps)
      qed
  next
    case PosZero
    assume bounds:0 ≤ rl1 ∧ rl2 ≤ 0 ∧ 0 ≤ ru2
    have r1:r1 ≥ 0 using bounds dual-order.trans grl1 by blast
    have bound:0 ≤ ru1 using r1 bounds grl1 grl2 gru1 gru2 dual-order.trans by
auto
    consider r2 ≥ 0 | r2 ≤ 0 using le-cases by auto
    then show ?thesis
    proof (cases)
      case 1
      have g1:rl1 * rl2 ≤ 0
        using r1 1 bounds grl1 grl2 gru1 gru2 mult-le-0-iff by blast
      have g2:0 ≤ r1 * r2
        using r1 1 bounds grl1 grl2 gru1 gru2 zero-le-mult-iff by blast
      from g1 and g2
      have up:rl1 * rl2 ≤ r1 * r2
        by auto
```

```
      show ?thesis
        using up maxU12 maxU34 bigMaxU wmax.elims min-repU2 min-repU1
        min.coboundedI1 min.commute maxt12 maxt34 repL-def timesll tl.simps
        by metis
    next
      case 2
      have g1:ru1 * rl2 ≤ ru1 * r2
        using r1 2 bounds bound grl1 grl2 gru1 gru2
        using mult-left-mono by blast
      have g2:ru1 * r2 ≤ r1 * r2
        using r1 2 bounds bound grl2 gru2
        by (metis mult-left-mono-neg gru1 mult.commute)
      from g1 and g2
      have up:ru1 * rl2 ≤ r1 * r2
        by auto
      show ?thesis
          using up maxU12 maxU34 bigMaxU wmin.elims min-repU2 min-repU1
maxt34
        by (metis repL-def timesul tl.simps)
      qed
  next
    case NegZero
    assume bounds:ru1 ≤ 0 ∧ rl2 ≤ 0 ∧ 0 ≤ ru2
    have r1:r1 ≤ 0  using bounds dual-order.trans gru1 by blast
    have bound:rl1 ≤ 0 using r1  bounds grl1 grl2 gru1 gru2 dual-order.trans by
auto
    consider r2 ≥ 0 | r2 ≤ 0 using le-cases by auto
    then show ?thesis
    proof (cases)
      case 1
      assume r2:r2 ≥ 0
      have g1:rl1 * ru2 ≤ rl1 * r2
        using r1 r2 bounds bound grl1 grl2 gru1 gru2
        by (metis mult-le-cancel-left leD)
      have g2:rl1 * r2 ≤ r1 * r2
        using r1 r2 bounds grl1 grl2 gru1 gru2 mult-right-mono
        by (simp add: mult-le-0-iff)
      from g1 and g2
      have up:rl1 * ru2 ≤ r1 * r2
        by auto
      show ?thesis
      using up maxU12 maxU34 bigMaxU min-repU2 min-repU1 min.coboundedI1
maxt34
      by (metis min-repU2 repL-def tl.simps)
    next
      case 2
      assume r2:r2 ≤ 0
      have lower:rl1 ≤ 0 using bounds dual-order.trans grl1 r1 by blast
      have g1:ru1 * ru2 ≤ 0
```

```
      using r1 r2 bounds  grl1 grl2 gru1 gru2 mult-le-0-iff by blast
    have g2:0 ≤ r1 * r2
      using r1 r2
      by (simp add: zero-le-mult-iff)
    from g1 and g2
    have up:ru1 * ru2 ≤ r1 * r2
      by auto
    show ?thesis
    using up maxU12 maxU34 bigMaxU wmin.elims min-repU2 min-repU1
      min.coboundedI1 min.commute maxt34
    by (metis repL-def tl.simps)
    qed
  next
    case NegNeg
    assume bounds:ru1 ≤ 0 ∧ ru2 ≤ 0
    have r1:r1 ≤ 0 using bounds dual-order.trans gru1 by blast
    have r2:r2 ≤ 0 using bounds dual-order.trans gru2 by blast
    have lower1:rl1 ≤ 0 using bounds dual-order.trans grl1 r1 by blast
    have lower2:rl2 ≤ 0 using bounds dual-order.trans grl2 r2 by blast
    have g1:ru1 * ru2 ≤ ru1 * r2
      using r1 r2 bounds  grl1 grl2 gru1 gru2
      using    not-less mult-le-cancel-left
      by metis
    have g2:ru1 * r2 ≤ r1 * r2
     using r1 r2 bounds grl1 grl2 gru1 gru2 mult-le-cancel-left mult.commute not-le
lower1 lower2
      by metis
    from g1 and g2
    have up:ru1 * ru2 ≤ r1 * r2
      by auto
    show ?thesis
      using up maxU12 maxU34 bigMaxU
      wmin.elims min-repU2 min-repU1
      min.coboundedI1 min.commute maxt34
      by (metis repL-def tl.simps)
  next
    case NegPos
    assume bounds:ru1 ≤ 0 ∧ 0 ≤ rl2
    have r1:r1 ≤ 0 using bounds dual-order.trans gru1 by blast
    have r2:r2 ≥ 0 using bounds dual-order.trans grl2 by blast
    have lower1:rl1 ≤ 0 using bounds dual-order.trans grl1 r1 by blast
    have lower2:rl2 ≥ 0 using bounds by auto
    have upper1:ru1 ≤ 0 using bounds by auto
    have upper2:ru2 ≥ 0 using bounds dual-order.trans gru2 r2 by blast
    have g1:rl1 * ru2 ≤ rl1 * r2
      using r1 r2 bounds grl1 grl2 gru1 gru2 less-le-trans upper1 lower2
      by (metis mult-le-cancel-left not-less)
    have g2:rl1 * r2 ≤ r1 * r2
      using r1 upper1 r2 mult-right-mono mult-le-0-iff grl1 by blast
```

from *g1* **and** *g2*
have *up*:*rl1* $*$ *ru2* $\leq$ *r1* $*$ *r2*
by *auto*
show *?thesis*
using *up maxU12 maxU34 bigMaxU wmin.elims min-repU2 min-repU1 maxt12 maxt34*
by (*metis repL-def timeslu tl.simps*)
next
case *PosNeg*
assume *bounds*:*0* $\leq$ *rl1* $\wedge$ *ru2* $\leq$ *0*
have *r1*:*r1* $\geq$ *0* **using** *bounds dual-order.trans grl1* **by** *blast*
have *r2*:*r2* $\leq$ *0* **using** *bounds dual-order.trans gru2* **by** *blast*
have *lower1*:*rl1* $\geq$ *0* **using** *bounds* **by** *auto*
have *lower2*:*rl2* $\leq$ *0* **using** *dual-order.trans grl2 r2* **by** *blast*
have *upper1*:*ru1* $\geq$ *0* **using** *dual-order.trans gru1 u1 r1* **by** *blast*
have *upper2*:*ru2* $\leq$ *0* **using** *bounds* **by** *auto*
have *g1*:*ru1* $*$ *rl2* $\leq$ *ru1* $*$ *r2*
using *r1 r2 bounds grl1 grl2 gru1 gru2 mult-left-mono less-le-trans not-less*
by *metis*
have *g2*:*ru1* $*$ *r2* $\leq$ *r1* $*$ *r2*
using *r1 lower1 r2 not-less gru2 gru1 grl1 grl2*
by (*metis mult-le-cancel-left mult.commute*)
from *g1* **and** *g2*
have *up*:*ru1* $*$ *rl2* $\leq$ *r1* $*$ *r2*
by *auto*
show *tl* l_1 u_1 l_2 u_2 $\equiv_L$ *r1* $*$ *r2*
using *up maxU12 maxU34 bigMaxU wmin.elims min-repU2 min-repU1*
by (*metis repL-def timesul tl.simps*)
next
case *PosPos*
assume *bounds*:*0* $\leq$ *rl1* $\wedge$ *0* $\leq$ *rl2*
have *r1*:*r1* $\geq$ *0* **using** *bounds dual-order.trans grl1* **by** *blast*
have *r2*:*r2* $\geq$ *0* **using** *bounds dual-order.trans grl2* **by** *blast*
have *lower1*:*rl1* $\geq$ *0* **using** *bounds* **by** *auto*
have *lower2*:*rl2* $\geq$ *0* **using** *bounds* **by** *auto*
have *upper1*:*ru1* $\geq$ *0* **using** *dual-order.trans gru1 u1 r1* **by** *blast*
have *upper2*:*ru2* $\geq$ *0* **using** *dual-order.trans gru2 u2 r2 bounds* **by** *blast*
have *g1*:*rl1* $*$ *rl2* $\leq$ *rl1* $*$ *r2*
using *r1 r2 bounds grl1 grl2 gru1 gru2*
using *mult-left-mono*
using *leD leI less-le-trans* **by** *auto*
have *g2*:*rl1* $*$ *r2* $\leq$ *r1* $*$ *r2*
using *r1 r2 bounds grl1 grl2 gru1 gru2*
using *mult-right-mono* **by** *blast*
from *g1* **and** *g2*
have *up*:*rl1* $*$ *rl2* $\leq$ *r1* $*$ *r2*
by *auto*
show *?thesis*
using *up maxU12 maxU34 bigMaxU min-repU2 min-repU1 min.coboundedI1*

maxt12 maxt34
    by (*metis repL-def tl.simps*)
  qed
qed

Most significant bit only changes under successor when all other bits are 1

lemma *msb-succ*:
  fixes *w* :: *32 Word.word*
  assumes *neq1*:*uint w* ≠ *0xFFFFFFFF*
  assumes *neq2*:*uint w* ≠ *0x7FFFFFFF*
  shows *msb* (*w* + *1*) = *msb w*
  proof −
    have *w* ≠ *0xFFFFFFFF*
      using *neq1* **by** *auto*
    then have *neqneg1*:*w* ≠ −*1* **by** *auto*
    have *w* ≠ *0x7FFFFFFF*
      using *neq2* **by** *auto*
    then have *neqneg2*:*w* ≠ (*2*^*31*)−*1* **by** *auto*
    show *?thesis* **using** *neq1 neq2* **unfolding** *msb-big*
    using *word-le-make-less*[*of w* + *1 0x80000000*]
        word-le-make-less[*of w 0x80000000*]
        neqneg1 neqneg2
      by *auto*
  qed

Negation commutes with msb except at edge cases

lemma *msb-non-min*:
  fixes *w* :: *32 Word.word*
  assumes *neq1*:*uint w* ≠ *0*
  assumes *neq2*:*uint w* ≠ ((*2*^(*len-of* (*TYPE*(*31*)))))
  shows *msb* (*uminus w*) = *HOL.Not*(*msb*(*w*))
  proof −
    have *fact1*:*uminus w* = *word-succ* (~~ *w*)
      by (*rule twos-complement*)
    have *fact2*:*msb* (~~*w*) = *HOL.Not*(*msb w*)
      using *word-ops-msb*[*of w*]
      by *auto*
    have *neqneg1*:*w* ≠ *0* **using** *neq1* **by** *auto*
    have *not-undef*:*w* ≠ *0x80000000*
      using *neq2* **by** *auto*
    then have *neqneg2*:*w* ≠ (*2*^*31*) **by** *auto*
    from ‹*w* ≠ *0*› **have** ‹~~ *w* ≠ ~~ *0*›
      by (*simp only*: *bit.compl-eq-compl-iff*) *simp*
    then have (~~ *w*) ≠ *0xFFFFFFFF*
      by *auto*
    then have *uintNeq1*:*uint* (~~ *w*) ≠ *0xFFFFFFFF*
      using *uint-distinct*[*of* ~~*w 0xFFFFFFFF*]
      by *auto*
    from ‹*w* ≠ *2* ^ *31*› **have** ‹~~ *w* ≠ ~~ *2* ^ *31*›

```
    by (simp only: bit.compl-eq-compl-iff) simp
  then have (~~ w) ≠ 0x7FFFFFFF
    by auto
  then have uintNeq2: uint (~~ w) ≠ 0x7FFFFFFF
    using uint-distinct[of ~~w 0x7FFFFFFF]
    by auto
  have fact3:msb ((~~w) + 1) = msb (~~w)
    apply(rule msb-succ[of ~~w])
    using neq1 neq2 uintNeq1 uintNeq2 by auto
  show msb (uminus w) = HOL.Not(msb(w))
    using fact1 fact2 fact3 by (simp add: word-succ-p1)
  qed
```

Only 0x80000000 preserves msb=1 under negation

```
lemma msb-min-neg:
  fixes w::word
  assumes msb1:msb (- w)
  assumes msb2:msb w
  shows uint w = ((2^(len-of (TYPE(31)))))
proof (rule ccontr)
  from ‹msb w› have ‹w ≠ 0›
    using word-msb-0 by auto
  then have ‹uint w ≠ 0›
    by transfer simp
  moreover assume ‹uint w ≠ 2 ^ LENGTH(31)›
  ultimately have ‹msb (- w) ⟷ ¬ msb w›
    by (rule msb-non-min)
  with assms show False
    by simp
qed
```

Only 0x00000000 preserves msb=0 under negation

```
lemma msb-zero:
  fixes w::word
  assumes msb1:¬ msb (- w)
  assumes msb2:¬ msb w
  shows uint w = 0
  proof -
    have neq:w ≠ ((2 ^ len-of TYPE(31))::word) using msb1 msb2 by auto
    have eq:uint ((2 ^ len-of TYPE(31))::word) = 2 ^ len-of TYPE(31)
      by auto
    then have neq:uint w ≠ uint ((2 ^ len-of TYPE(31))::word)
      using uint-distinct[of w 2^len-of TYPE(31)] neq eq by auto
    show ?thesis
      using msb1 msb2 minus-zero msb-non-min[of w] neq by force
  qed
```

Finite numbers alternate msb under negation

```
lemma msb-pos:
```

```
  fixes w::word
  assumes msb1:msb (− w)
  assumes msb2:¬ msb w
  shows uint w ∈ {1 .. (2^((len-of TYPE(32)) − 1))−1}
  proof −
    have main: w ∈ {1 .. (2^((len-of TYPE(32)) − 1))−1}
      using msb1 msb2 apply(clarsimp)
      unfolding word-msb-sint
      apply(rule conjI)
      apply (metis neg-equal-0-iff-equal not-le word-less-1)
      proof −
        have imp:w ≥ 0x80000000 ⟹ False
          proof −
            assume geq:w ≥ 0x80000000
            then have msb w
              using msb-big [of w] by auto
            then show False using msb2 by auto
          qed
        have mylem:⋀w1 w2::word. uint w1 ≥ uint w2 ⟹ w1 ≥ w2
          subgoal for w1 w2
            by (simp add: word-le-def)
          done
        have mylem2:⋀w1 w2::word.  w1 >  w2 ⟹ uint w1 > uint w2
          subgoal for w1 w2
            by (simp add: word-less-def)
          done
        have gr-to-geq:w > 0x7FFFFFFF ⟹ w ≥ 0x80000000
          apply(rule mylem)
          using mylem2[of 0x7FFFFFFF w] by auto
        have taut:w ≤ 0x7FFFFFFF ∨ w > 0x7FFFFFFF by auto
        then show w ≤ 0x7FFFFFFF
        using imp taut gr-to-geq by auto
      qed
    have set-eq:(uint ‘ (({1..(minus(2 ^ (minus(len-of TYPE(32))  1)) 1)}))::word
set))
              = ({1..minus(2 ^ (minus (len-of TYPE(32)) 1)) 1}::int set)
      apply(auto simp add: word-le-def)
      subgoal for xa
      proof −
        assume lower:1 ≤ xa and upper:xa ≤ 2147483647
        then have in-range:xa ∈ {0 .. 2^32−1} by auto
        then have xa ∈ range (uint::word ⇒ int)
          unfolding word-uint.Rep-range uints-num by auto
        then obtain w::word  where xaw:xa = uint w by auto
        then have w ∈ {1..0x7FFFFFFF}
        using lower upper apply(clarsimp, auto)
        by (auto simp add: word-le-def)
        then show ?thesis
          using uint-distinct uint-distinct main image-eqI word-le-def xaw by blast
```

```
    qed
    done
  then show  uint w ∈ {1..2 ^ (len-of TYPE(32) − 1) − 1}
    using uint-distinct uint-distinct main image-eqI
    by blast
  qed

lemma msb-neg:
  fixes w::word
  assumes msb1:¬ msb (− w)
  assumes msb2:msb w
  shows uint w ∈ {2^((len-of TYPE(32) − 1))+1 .. 2^((len-of TYPE(32)))−1}
  proof −
    have mylem:⋀w1 w2::word. uint w1 ≥ uint w2 ⟹ w1 ≥ w2
      by (simp add: word-le-def)
    have mylem2:⋀w1 w2::word.  w1 >  w2 ⟹ uint w1 > uint w2
      by (simp add: word-less-def)
    have gr-to-geq:w > 0x80000000 ⟹ w ≥ 0x80000001
      apply(rule mylem)
      using mylem2[of 0x80000000 w] by auto
    have taut:w ≤ 0x80000000 ∨ 0x80000000 < w by auto
    have imp:w ≤ 0x80000000 ⟹ False
      proof −
        assume geq:w ≤ 0x80000000
        then have (msb (−w))
          using msb-big [of − w] msb-big [of w]
          by (simp add: msb2)
        then show False using msb1 by auto
      qed
   have main: w ∈ {2^((len-of TYPE(32)) − 1)+1 .. 2^((len-of TYPE(32)))−1}

      using msb1 msb2 apply(clarsimp)
      unfolding word-msb-sint
      proof −
        show 0x80000001 ≤ w
        using imp taut gr-to-geq by auto
      qed
   have set-eq:(uint ‘ (({2^((len-of TYPE(32) − 1))+1 .. 2^((len-of TYPE(32)))−1}::word
set))
                  = {2^((len-of TYPE(32) − 1))+1 .. 2^((len-of TYPE(32)))−1}
      apply(auto)
      subgoal for xa by (simp add: word-le-def)
      subgoal for w using uint-lt [of w] by simp
      subgoal for xa
      proof −
        assume lower:2147483649 ≤ xa and upper:xa ≤ 4294967295
        then have in-range:xa ∈ {0x80000000 .. 0xFFFFFFFF} by auto
        then have xa ∈ range (uint::word ⇒ int)
          unfolding word-uint.Rep-range uints-num by auto
```

then obtain *w*::*word* **where** *xaw*:*xa* = *uint w* **by** *auto*
then have *the-in*:*w* ∈ {*0x80000001* .. *0xFFFFFFFF*}
using *lower upper*
by (*auto simp add*: *word-le-def*)
have *the-eq*:(*0xFFFFFFFF*::*word*) = −*1* **by** *auto*
from *the-in the-eq* **have** *w* ∈ {*0x80000001* .. −*1*} **by** *auto*
then show *?thesis*
using *uint-distinct uint-distinct main image-eqI word-le-def xaw* **by** *blast*
qed
done
then show *uint w* ∈ {*2*^((*len-of TYPE*(*32*)) − *1*)+*1* .. *2*^((*len-of TYPE*(*32*)))−*1*}

using *uint-distinct uint-distinct main image-eqI*
by *blast*
qed

2s-complement commutes with negation except edge cases

lemma *sint-neg-hom*:
fixes *w* :: *32 Word.word*
shows *uint w* ≠ ((*2*^(*len-of* (*TYPE*(*31*))))) ⟹ (*sint*(−*w*) = −(*sint w*))
unfolding *word-sint-msb-eq* **apply** *auto*
subgoal using *msb-min-neg* **by** *auto*
prefer *3* **subgoal using** *msb-zero*[*of w*] **by** (*simp add*: *msb-zero*)
proof −
assume *msb1*:*msb* (− *w*)
assume *msb2*:¬ *msb w*
have *uint w* ∈ {*1* .. (*2*^((*len-of TYPE*(*32*)) − *1*))−*1*} **using** *msb-pos*[*OF msb1 msb2*] **by** *auto*
then have *bound*:*uint w* ∈ {*1* .. *0x7FFFFFFF*} **by** *auto*
have *size*:*size* (*w*::*32 Word.word*) = *32* **using** *Word.word-size*[*of w*] **by** *auto*
have *lem*:⋀*x*::*int*. ⋀*n*::*nat*. *x* ∈ {*1*..(*2*^*n*)−*1*} ⟹ ((− *x*) *mod* (*2*^*n*)) − (*2*^*n*) = − *x*
subgoal for *x n*
apply(*cases x mod 2*^*n* = *0*)
by(*auto simp add*: *zmod-zminus1-eq-if*[*of x 2*^*n*])
done
have *lem-rule*:*uint w* ∈ {*1..2* ^ *32* − *1*}
⟹ (− *uint w mod 4294967296*) − *4294967296* = − *uint w*
using *lem*[*of uint w 32*] **by** *auto*
have *almost*:− *uint w mod 4294967296* − *4294967296* = − *uint w*
apply(*rule lem-rule*)
using *bound* **by** *auto*
show *uint* (− *w*) − *2* ^ *size* (− *w*) = − *uint w*
using *bound*
unfolding *Word.uint-word-ariths word-size-neg* **by** (*auto simp add*: *size almost*)
next
assume *neq*:*uint w* ≠ *0x80000000*
assume *msb1*:¬ *msb* (− *w*)
assume *msb2*:*msb w*

have *bound*:*uint w* ∈ {*0x80000001*.. *0xFFFFFFFF*} **using** *msb1 msb2 msb-neg* **by** *auto*
have *size*:*size* (*w*::*32 Word.word*) = *32* **using** *Word.word-size*[*of w*] **by** *auto*
have *lem*:⋀*x*::*int*. ⋀*n*::*nat*. *x* ∈ {*1*..(*2*^*n*)−*1*} ⟹ (−*x mod* (*2*^*n*)) = (*2*^*n*) − *x*
subgoal for *x n*
apply(*auto*)
apply(*cases x mod 2*^*n* = *0*)
by (*simp add*: *zmod-zminus1-eq-if*[*of x 2*^*n*])+
done
from *bound*
have *wLeq*: *uint w* ≤ *4294967295*
and *wGeq*: *2147483649* ≤ *uint w*
by *auto*
from *wLeq* **have** *wLeq'*:*uint w* ≤ *4294967296* **by** *fastforce*
have *f3*: (*0* ≤ *4294967296* + − *1* ∗ *uint w* + − *1* ∗ ((*4294967296* + − *1* ∗ *uint w*) *mod 4294967296*))
= (*uint w* + (*4294967296* + − *1* ∗ *uint w*) *mod 4294967296* ≤ *4294967296*)
by *auto*
have *f4*: (*0* ≤ *4294967296* + − *1* ∗ *uint w*) = (*uint w* ≤ *4294967296*)
by *auto*
have *f5*: ∀ *i ia*. ¬ (*0*::*int*) ≤ *i* ∨ *0* ≤ *i* + − *1* ∗ (*i mod ia*)
by (*simp add*: *zmod-le-nonneg-dividend*)
then have *f6*: *uint w* + (*4294967296* + − *1* ∗ *uint w*) *mod 4294967296* ≤ *4294967296*
using *f4 f3 wLeq'* **by** *blast*
have *f7*: *4294967296* + − *1* ∗ *uint w* + − *4294967296* = − *1* ∗ *uint w*
by *auto*
have *f8*: − (*1*::*int*) ∗ *4294967296* = − *4294967296*
by *auto*
have *f9*: (*0* ≤ − *1* ∗ *uint w*) = (*uint w* ≤ *0*)
by *auto*
have *f10*: (*4294967296* + *−1* ∗ *uint w* + *−1* ∗ ((*4294967296* + *−1* ∗ *uint w*) *mod 4294967296*) ≤ *0*)
= (*4294967296* ≤ *uint w* + (*4294967296* + − *1* ∗ *uint w*) *mod 4294967296*)
by *auto*
have *f11*: ¬ *4294967296* ≤ (*0*::*int*)
by *auto*
have *f12*: ∀ *x0*. ((*0*::*int*) < *x0*) = (¬ *x0* ≤ *0*)
by *auto*
have *f13*: ∀ *x0 x1*. ((*x1*::*int*) < *x0*) = (¬ *0* ≤ *x1* + − *1* ∗ *x0*)
by *auto*
have *f14*: ∀ *x0 x1*. ((*x1*::*int*) ≤ *x1 mod x0*) = (*x1* + − *1* ∗ (*x1 mod x0*) ≤ *0*)
by *auto*
have ¬ *uint w* ≤ *0*
using *wGeq* **by** *fastforce*
then have *4294967296* ≤ *uint w* + (*4294967296* + − *1* ∗ *uint w*) *mod 4294967296*
using *f14 f13 f12 f11 f10 f9 f8 f7* **by** (*metis* (*no-types*) *int-mod-ge*)
then

show *uint* (− *w*) = 2 ^ *size w* − *uint w*
using *f6*
unfolding *Word.uint-word-ariths*
by (*auto simp add*: *size f4*)
qed

2s-complement encoding is injective

lemma *sint-dist*:
fixes *x y* ::*word*
assumes *x* ≠ *y*
shows *sint x* ≠ *sint y*
by (*simp add*: *assms*)

3.9 Negation

fun *wneg* :: *word* ⇒ *word*
where *wneg w* =
(*if w* = *NEG-INF then POS-INF else if w* = *POS-INF then NEG-INF else* −*w*)

word negation is correct

lemma *wneg-lemma*:
assumes *eq*:*w* ≡$_E$ (*r*::*real*)
shows *wneg w* ≡$_E$ −*r*
apply(*rule repe.cases*[*OF eq*])
apply(*auto intro*!: *repNEG-INF repPOS-INF simp add*: *repe.simps*)[*2*]
subgoal for *ra*
proof −
assume *eq*:*w* = *ra*
assume *i*:*r* = (*real-of-int* (*sint ra*))
assume *bounda*: (*real-of-int* (*sint ra*)) < (*real-of-int* (*sint POS-INF*))
assume *boundb*: (*real-of-int* (*sint NEG-INF*)) < (*real-of-int* (*sint ra*))
have *raNeq*:*ra* ≠ *2147483647*
using *sint-range*[*OF bounda boundb*]
by (*auto*)
have *raNeqUndef*:*ra* ≠ *2147483648*
using *int-not-undef*[*OF bounda boundb*]
by (*auto*)
have *uint ra* ≠ *uint* ((*2* ^ *len-of TYPE*(*31*))::*word*)
apply (*rule uint-distinct*)
using *raNeqUndef* **by** *auto*
then have *raNeqUndefUint*:*uint ra* ≠ ((*2* ^ *len-of TYPE*(*31*)))
by *auto*
have *res1*:*wneg w* ≡$_E$ (*real-of-int* (*sint* (*wneg w*)))
apply (*rule repINT*)
using *sint-range*[*OF bounda boundb*] *sint-neg-hom*[*of ra*, *OF raNeqUndefUint*]
raNeq raNeqUndefUint raNeqUndef eq
by(*auto*)
have *res2*:− *r* = (*real-of-int* (*sint* (*wneg w*)))

using *eq bounda boundb i sint-neg-hom*[*of ra*, *OF raNeqUndefUint*] *raNeq raNeqUndef eq*
apply *auto*
apply *transfer*
apply *simp*
done
show *?thesis*
using *res1 res2* **by** *auto*
qed
done

3.10 Comparison

fun *wgreater* :: *word* ⇒ *word* ⇒ *bool*
where *wgreater w1 w2* = (*sint w1* > *sint w2*)

lemma *neg-less-contra*:⋀*x*. *Suc x* < − (*Suc x*) ⟹ *False*
by *auto*

Comparison < is correct

lemma *wgreater-lemma*:*w1* ≡$_L$ (*r1*::*real*) ⟹ *w2* ≡$_U$ *r2* ⟹ *wgreater w1 w2* ⟹ *r1* > *r2*
proof (*auto simp add*: *repU-def repL-def*)
fix r'_1 r'_2
assume *sint-le*:*sint w1* > *sint w2*
then have *sless*:(*w2* <*s w1*) **using** *word-sless-alt* **by** *auto*
assume *r1-leq*:r'_1 ≤ *r1*
assume *r2-leq*:*r2* ≤ r'_2
assume *wr1*:*w1* ≡$_E$ r'_1
assume *wr2*:*w2* ≡$_E$ r'_2
have *greater*:r'_1 > r'_2
using *wr1 wr2* **apply**(*auto simp add*: *repe.simps*)
prefer *4* **using** *sless sint-le*
apply (*auto simp add*: *less-le-trans not-le*)
apply *transfer* **apply** *simp*
apply *transfer* **apply** *simp*
apply *transfer* **apply** *simp*
done
show *r1* > *r2*
using *r1-leq r2-leq greater* **by** *auto*
qed

Comparison ≥ of words

fun *wgeq* :: *word* ⇒ *word* ⇒ *bool*
where *wgeq w1 w2* =
((¬ ((*w2* = *NEG-INF* ∧ *w1* = *NEG-INF*)
∨(*w2* = *POS-INF* ∧ *w1* = *POS-INF*))) ∧
(*sint w2* ≤ *sint w1*))

Comparison ≥ of words is correct

lemma *wgeq-lemma*:*w1* $\equiv_L$ *r1* $\Longrightarrow$ *w2* $\equiv_U$ (*r2*::*real*) $\Longrightarrow$ *wgeq w1 w2* $\Longrightarrow$ *r1* $\geq$ *r2*
proof (*unfold wgeq.simps*)
assume *assms*:$\neg$ (*w2* = *NEG-INF* $\wedge$ *w1* = *NEG-INF* $\vee$ *w2* = *POS-INF* $\wedge$ *w1* = *POS-INF*) $\wedge$ *sint w2* $\leq$ *sint w1*
assume *a1*:*w1* $\equiv_L$ *r1* **and** *a2*:*w2* $\equiv_U$ (*r2*::*real*)
from *assms* **have** *sint-le*:*sint w2* $\leq$ *sint w1* **by** *auto*
then have *sless*:(*w2* <=*s w1*) **using** *word-sless-alt word-sle-def* **by** *auto*
obtain r'_1 r'_2 **where** *r1-leq*:$r'_1 \leq$ *r1* **and** *r2-leq*:*r2* $\leq r'_2$
and *wr1*:*w1* $\equiv_E r'_1$ **and** *wr2*:*w2* $\equiv_E r'_2$
using *a1 a2* **unfolding** *repU-def repL-def* **by** *auto*
from *assms* **have** *check1*:$\neg$ (*w1* = *NEG-INF* $\wedge$ *w2* = *NEG-INF*) **by** *auto*
from *assms* **have** *check2*:$\neg$ (*w1* = *POS-INF* $\wedge$ *w2* = *POS-INF*) **by** *auto*
have *less*:$r'_2 \leq r'_1$
using *sless sint-le check1 check2 repe.simps wr2 wr1*
apply (*auto simp add*: *repe.simps*)
apply *transfer*
apply *simp*
apply *transfer*
apply *simp*
apply *transfer*
apply *simp*
apply *transfer*
apply *simp*
apply *transfer*
apply *simp*
apply *transfer*
apply *simp*
apply *transfer*
apply *simp*
done
show *r1* $\geq$ *r2*
using *r1-leq r2-leq less* **by** *auto*
qed

3.11 Absolute value

Absolute value of word

fun *wabs* :: *word* $\Rightarrow$ *word*
where *wabs l1* = (*wmax l1* (*wneg l1*))

Correctness of wmax

lemma *wabs-lemma*:
assumes *eq*:*w* $\equiv_E$ (*r*::*real*)
shows *wabs w* $\equiv_E$ (*abs r*)
proof −

have *w*:*wmax w* (*wneg w*) ≡$_E$ *max r* (−*r*) **by** (*rule wmax-lemma*[*OF eq wneg-lemma*[*OF eq*]])
have *r*:*max r* (−*r*) = *abs r* **by** *auto*
from *w r* **show** *?thesis* **by** *auto*
qed

declare *more-real-of-word-simps* [*simp del*]

end

4 Finite Strings

Finite-String.thy implements a type of strings whose lengths are bounded by a constant defined at "proof-time", by taking a sub-type of the built-in string type. A finite length bound is important for applications in real analysis, specifically the Differential-Dynamic-Logic (dL) entry, because finite-string identifiers are used as the index of a real vector, only forming a Euclidean space if identifiers are finite.

We include finite strings in this AFP entry both to promote using it as the basis of future versions of the dL entry and simply in case the typeclass instances herein are useful. One could imagine using this type in file formats with fixed-length fields.

theory *Finite-String*
imports
Main
HOL−Library.Char-ord
begin

This theory uses induction on pairs of lists often: give names to the cases

lemmas *list-induct2'*[*case-names BothNil LeftCons RightCons BothCons*] = *List.list-induct2'*

Set a hard-coded global maximum string length

definition *max-str*:*MAX-STR* = *20*

Finite strings are strings whose size is within the maximum

typedef *fin-string* = {*s*::*string*. *size s* ≤ *MAX-STR*}
morphisms *Rep-fin-string Abs-fin-string*
apply(*auto*)
apply(*rule exI*[**where** *x*=*Nil*])
by(*auto simp add*: *max-str*)

Lift definition of string length

setup-lifting *Finite-String.fin-string.type-definition-fin-string*
lift-definition *ilength*::*fin-string* ⇒ *nat* **is** *length* **done**

Product of types never decreases cardinality

lemma *card-prod-finite*:
fixes *C*:: *char set* **and** *S*::*string set*
assumes *C*:*card C* ≥ *1* **and** *S*:*card S* ≥ *0*
shows *card C* ∗ *card S* ≥ *card S*
using *C S* **by** *auto*

fun *cons* :: (*'a* ∗ *'a list*) ⇒ *'a list*
where *cons* (*x*,*y*) = *x* # *y*

Finite strings are finite

instantiation *fin-string* :: *finite* **begin**
instance proof
have *any*:∀ *i*::*nat*. *card* {*s*::*string*. *length s* ≤ *i*} > *0*
apply(*auto*)
subgoal for *i*
proof (*induct i*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc k*)
assume *IH*:*card* {*s*::*string*. *length s* ≤ *k*} > *0*
let *?c* = (*UNIV*::*char set*)
let *?ih* = {*s*::*string*. *length s* ≤ *k*}
let *?prod* = (*?c* × *?ih*)
let *?b* = (*cons* ‘ *?prod*)
let *?A* = {*s*::*string*. *length s* ≤ *Suc k*}
let *?B* = *insert* [] *?b*
have *IHfin*:*finite ?ih* **using** *IH card-ge-0-finite* **by** *blast*
have *finChar*:*finite ?c* **using** *card-ge-0-finite finite-code* **by** *blast*
have *finiteProd*:*finite ?prod*
using *Groups-Big.card-cartesian-product IHfin finChar* **by** *auto*
have *cardCons*:*card ?b* = *card ?prod*
apply(*rule Finite-Set.card-image*)
by(*auto simp add*: *inj-on-def*)
have *finiteCons*:*finite ?b* **using** *cardCons finiteProd card-ge-0-finite* **by** *blast*
have *finiteB*:*finite ?B* **using** *finite-insert finiteCons* **by** *auto*
have *lr*:⋀*x*. *x* ∈ *?A* ⟹ *x* ∈ *?B* **subgoal for** *x*
apply(*auto*) **apply**(*cases x*) **apply** *auto*
by (*metis UNIV-I cons.simps image-eqI mem-Collect-eq mem-Sigma-iff*) **done**
have *rl*:⋀*x*. *x* ∈ *?B* ⟹ *x* ∈ *?A* **subgoal for** *x*
by(*auto*) **done**
have *isCons*:*?A* = *?B*
using *lr rl* **by** *auto*
show *?case*
using *finiteB isCons IH* **by** (*simp add*: *card.insert-remove*)
qed
done
note *finMax* = *card-ge-0-finite*[*OF spec*[*OF any*, *of MAX-STR*]]
have *fin*:*finite* {*x* | *x y* . *x* = *Abs-fin-string y* ∧ *y* ∈ {*s*. *length s* ≤ *MAX-STR*}}

using *Abs-fin-string-cases finMax* **by** *auto*
have *univEq*:*UNIV* = {*x* | *x y* . *x* = *Abs-fin-string y* ∧ *y* ∈ {*s*. *length s* ≤ *MAX-STR*}}
using *Abs-fin-string-cases*
by (*metis* (*mono-tags*, *lifting*) *Collect-cong UNIV-I top-empty-eq top-set-def*)
then have *finite* (*UNIV* :: *fin-string set*) **using** *univEq fin* **by** *auto*
then show *finite* (*UNIV*::*fin-string set*) **by** *auto*
qed
end

Finite strings are linearly ordered, lexicographically

instantiation *fin-string* :: *linorder* **begin**
fun *lleq-charlist* :: *char list* ⇒ *char list* ⇒ *bool*
where
lleq-charlist Nil Nil = *True*
| *lleq-charlist Nil* - = *True*
| *lleq-charlist* - *Nil* = *False*
| *lleq-charlist* (*x* # *xs*)(*y* # *ys*) =
(*if x* = *y then lleq-charlist xs ys else x* < *y*)

fun *less-charlist* :: *char list* ⇒ *char list* ⇒ *bool*
where
less-charlist Nil Nil = *False*
| *less-charlist Nil* - = *True*
| *less-charlist* - *Nil* = *False*
| *less-charlist* (*x* # *xs*)(*y* # *ys*) =
(*if x* = *y then less-charlist xs ys else x* < *y*)

lift-definition *less-eq-fin-string*::*fin-string* ⇒ *fin-string* ⇒ *bool* **is** *lleq-charlist* **done**
lift-definition *less-fin-string*::*fin-string* ⇒ *fin-string* ⇒ *bool* **is** *less-charlist* **done**

lemma *lleq-head*:
fixes *L1 L2 x*
assumes *a*:
(⋀*z*. *lleq-charlist L2 z* ⟹ *lleq-charlist L1 z*)
lleq-charlist L1 L2
lleq-charlist (*x* # *L2*) *w*
shows *lleq-charlist* (*x* # *L1*) *w*
using *a* **by**(*induction arbitrary*: *x rule*: *List.list-induct2′*, *auto*)

lemma *lleq-less*:
fixes *x y*
shows (*less-charlist x y*) = (*lleq-charlist x y* ∧ ¬ *lleq-charlist y x*)
by(*induction rule*: *List.list-induct2′*, *auto*)

lemma *lleq-refl*:
fixes *x*
shows *lleq-charlist x x*
by(*induction x*, *auto*)

lemma *lleq-trans*:
fixes *x y z*
shows *lleq-charlist x y* $\Longrightarrow$ *lleq-charlist y z* $\Longrightarrow$ *lleq-charlist x z*
proof(*induction arbitrary*: *z rule*: *list-induct2'*)
case *BothNil*
then show *?case* **by** *auto*
next
case (*LeftCons x xs*)
then show *?case*
apply(*induction y*)
using *lleq-charlist.elims*(*2*) *lleq-charlist.simps*(*2*) **by** *blast*+
next
case (*RightCons y ys*)
then show *?case* **by** *auto*
next
case (*BothCons x xs y ys z*)
then show *?case*
using *lleq-head*[*of xs ys x z*] **apply**(*cases x* = *y*, *auto*)
apply(*cases z*, *auto*)
subgoal for *a list*
by(*cases x* = *a*, *auto*)
done
qed

lemma *lleq-antisym*:
fixes *x y*
shows *lleq-charlist x y* $\Longrightarrow$ *lleq-charlist y x* $\Longrightarrow$ *x* = *y*
proof(*induction rule*: *list-induct2'*)
case (*LeftCons x xs*) **then show** *?case* **by**(*cases xs*=*y*,*auto*)
next case (*RightCons y ys*) **then show** *?case* **by**(*cases x*=*ys*,*auto*)
next case (*BothCons x xs y ys*) **then show** *?case* **by**(*cases x*=*y*, *auto*)
qed (*auto*)

lemma *lleq-dichotomy*:
fixes *x y*
shows *lleq-charlist x y* $\vee$ *lleq-charlist y x*
by(*induction rule*: *List.list-induct2'*,*auto*)

instance
apply(*standard*)
unfolding *less-eq-fin-string-def less-fin-string-def*
apply (*auto simp add*: *lleq-less lleq-refl lleq-trans lleq-dichotomy*)
using *lleq-antisym less-eq-fin-string-def less-fin-string-def Rep-fin-string-inject* **by** *blast*
end

fun *string-expose*::*string* $\Rightarrow$ (*unit* + (*char* $*$ *string*))
where *string-expose Nil* = *Inl* ()

| *string-expose* (*c*#*cs*) = *Inr*(*c*,*cs*)

fun *string-cons*::*char* ⇒ *string* ⇒ *string*
where *string-cons c s* = (*if length s* ≥ *MAX-STR then s else c* # *s*)

lift-definition *fin-string-empty*::*fin-string* **is** *''''* **by**(*auto simp add*: *max-str*)
lift-definition *fin-string-cons*::*char* ⇒ *fin-string* ⇒ *fin-string* **is** *string-cons* **by** *auto*
lift-definition *fin-string-expose*::*fin-string* ⇒ (*unit* + (*char*∗*fin-string*)) **is** *string-expose*

apply(*auto simp add*: *dual-order.trans less-imp-le pred-sum.simps string-expose.elims*)
by (*metis dual-order.trans impossible-Cons le-cases string-expose.elims*)

Helper functions for enum typeclass instance

fun *fin-string-upto* :: *nat* ⇒ *fin-string list*
where
fin-string-upto 0 = [*fin-string-empty*]
| *fin-string-upto* (*Suc k*) =
(*let r* = *fin-string-upto k in*
let ab = (*enum-class.enum*::*char list*) *in*
fin-string-empty # *concat* (*map* (λ *c*. *map* (λ*s*. *fin-string-cons c s*) *r*) *ab*))

lemma *mem-appL*:*List.member L1 x* ⟹ *List.member* (*L1* @ *L2*) *x*
apply(*induction L1 arbitrary*: *L2*)
by(*auto simp add*:)

lemma *mem-appR*:*List.member L2 x* ⟹ *List.member* (*L1* @ *L2*) *x*
apply(*induction L1 arbitrary*: *L2*)
by(*auto simp add*:)

lemma *mem-app-or*:*List.member* (*L1* @ *L2*) *x* = *List.member L1 x* ∨ *List.member L2 x*
by *auto*

lemma *fin-string-nil*:
fixes *n*
shows *List.member* (*fin-string-upto n*) *fin-string-empty*
by(*induction n*, *auto simp add*: *fin-string-empty-def*)

List of every string. Not practical for code generation but used to show strings are an enum

definition *vals-def*[*code*]:*vals* ≡ *fin-string-upto MAX-STR*

definition *fin-string-enum* :: *fin-string list*
where *fin-string-enum* = *vals*
definition *fin-string-enum-all* :: (*fin-string* ⇒ *bool*) ⇒ *bool*
where *fin-string-enum-all* = (λ *f*. *list-all f vals*)
definition *fin-string-enum-ex* :: (*fin-string* ⇒ *bool*) ⇒ *bool*
where *fin-string-enum-ex* = (λ *f*. *list-ex f vals*)

Induct on the length of a bounded list, with access to index of element

lemma *length-induct*:
fixes *P*
assumes *len*:*length L* ≤ *MAX-STR*
assumes *BC*:*P* [] *0*
assumes *IS*:(⋀*k x xs*. *P xs k* ⟹ *P* ((*x* # *xs*)) (*Suc k*))
shows *P L* (*length L*)
proof −
have ⋀*k*. *length L* = *k* ⟹ *k* ≤ *MAX-STR* ⟹ *P L* (*length L*)
proof (*induction L*)
case *Nil* **then show** *?case* **using** *BC* **by** *auto*
next
case (*Cons a L*)
then have *it*:*P* (*L*) (*length L*) **using** *less-imp-le* **by** *fastforce*
then show *?case* **using** *IS*[*OF it*, *of a*] **by** (*auto*)
qed
then show *?thesis* **using** *BC IS len* **by** *auto*
qed

Induct on length of fin-string

lemma *ilength-induct*:
fixes *P*
assumes *BC*:*P fin-string-empty 0*
assumes *IS*:(⋀*k x xs*. *P xs k* ⟹ *P* (*Abs-fin-string* (*x* # *Rep-fin-string xs*)) (*Suc k*))
shows *P L* (*ilength L*)
apply(*cases L*)
apply(*unfold ilength-def*)
apply(*auto simp add*: *Abs-fin-string-inverse*)
subgoal for *y*
proof −
assume *a1*:*L* = *Abs-fin-string y*
assume *a2*: *length y* ≤ *MAX-STR*
have *main*:⋀*k*. *L* = *Abs-fin-string y* ⟹ *length y* = *k* ⟹ *k* ≤ *MAX-STR*
⟹ *P* (*Abs-fin-string y*) (*length y*)
subgoal for *k*
apply(*induction y arbitrary*: *k L*)
subgoal for *k* **using** *BC* **unfolding** *fin-string-empty-def* **by** *auto*
subgoal for *a y k L*
proof −
assume *IH*:(⋀*k L*. *L* = *Abs-fin-string y* ⟹ *length y* = *k* ⟹ *k* ≤ *MAX-STR*
⟹ *P* (*Abs-fin-string y*) (*length y*))
assume *L*:*L* = *Abs-fin-string* (*a* # *y*)
assume *l*:*length* (*a* # *y*) = *k*
assume *str*:*k* ≤ *MAX-STR*
have *yLen*:*length y* < *MAX-STR* **using** *l str* **by** *auto*
have *it*:*P* (*Abs-fin-string y*) (*length y*)
using *IH*[*of Abs-fin-string y k−1*, *OF refl*] **using** *L l str* **by** *auto*

```
        show  P (Abs-fin-string (a # y)) (length (a # y))
              using IS[OF it, of a] apply (auto simp add: fin-string-cons-def
Abs-fin-string-inverse)
          apply(cases MAX-STR ≤ length (Rep-fin-string (Abs-fin-string y)))
          using yLen by(auto simp add: l yLen Abs-fin-string-inverse)
       qed
       done
     done
   show ?thesis
     apply(rule main)
     using BC IS a1 a2 by auto
 qed
 done

lemma enum-chars:set (enum-class.enum::char list)= UNIV
 using Enum.enum-class.enum-UNIV by auto

lemma member-concat:List.member (concat LL) x = (∃ L. List.member LL L ∧
List.member L x)
 by auto
```

fin-string-upto k enumerates all strings up to length $min(k, MAX_STR)$

```
lemma fin-string-length:
 fixes L::string
 assumes len:length L ≤ k
 assumes Len:length L ≤ MAX-STR
 shows List.member (fin-string-upto k) (Abs-fin-string L)
proof −
 have BC:∀ j≥0. 0 ≤ MAX-STR ⟶ length [] = 0 ⟶
   List.member (fin-string-upto j) (Abs-fin-string [])
   apply(auto)
   subgoal for j
     apply(cases j)
     by (auto simp add: fin-string-empty-def)
   done
 have IS:(⋀k x xs.
      ∀ j≥k. k≤MAX-STR ⟶ length xs=k ⟶ List.member (fin-string-upto j)
(Abs-fin-string xs)⟹
     ∀ j≥Suc k. Suc k ≤ MAX-STR ⟶ length (x # xs) = Suc k
       ⟶ List.member (fin-string-upto j) (Abs-fin-string (x # xs)))
   subgoal for k x xs
   proof −
     assume ∀ j≥k. k ≤ MAX-STR ⟶ length xs = k
       ⟶ List.member (fin-string-upto j) (Abs-fin-string xs)
     then have IH:⋀j. j≥ k ⟹ k ≤ MAX-STR ⟹ length xs = k
       ⟹ List.member (fin-string-upto j) (Abs-fin-string xs)
       by auto
     show ?thesis
       apply(auto)
```

```
      subgoal for j
      proof −
        assume kj:Suc (length xs) ≤ j
        assume sucMax:Suc (length xs) ≤ MAX-STR
        assume ilen: k = length xs
        obtain jj where jj[simp]:j = Suc jj using kj Suc-le-D by auto
        then have kMax:k < MAX-STR using jj kj Suc-le-D ilen
          by (simp add: less-eq-Suc-le sucMax)
         have res:List.member (fin-string-upto (jj)) (Abs-fin-string xs)
          using IH[of jj] kj jj ilen  Suc-leD sucMax by blast
        have neq:Abs-fin-string [] ≠ Abs-fin-string (x # xs)
        using Abs-fin-string-inverse fin-string-empty.abs-eq fin-string-empty.rep-eq

            len length-Cons list.distinct(1) mem-Collect-eq
          by (metis ilen sucMax)
         have univ: set enum-class.enum  = (UNIV::char set) using enum-chars
by auto
        have List.member (fin-string-upto j) (Abs-fin-string (x # xs))
          using len sucMax res Abs-fin-string-inverse
        apply (auto simp add: fin-string-empty-def fin-string-cons-def enum-UNIV
image-iff not-le)
          apply fastforce
          done
        then show ‹Abs-fin-string (x # xs) ∈ set (fin-string-upto j)›
          by simp
      qed
      done
    qed
    done
  have impl:length L ≤ k ⟹ List.member (fin-string-upto k) (Abs-fin-string L)
    using len Len
    length-induct[where P = (λ L k. ∀ j ≥ k. k ≤ MAX-STR ⟶ length L = k
        ⟶ List.member (fin-string-upto j) (Abs-fin-string L))
      , OF Len BC IS]
      by auto
    show ?thesis
    using impl len by auto
qed


lemma fin-string-upto-length:
  shows List.member (fin-string-upto n) L ⟹ ilength L ≤ n
  apply(induction n arbitrary: L)
   apply(auto simp add: fin-string-empty-def Let-def ilength-def fin-string-cons-def

     Rep-fin-string-inverse Abs-fin-string-inverse enum-UNIV)
  apply fastforce
  done
```

fin-string-upto produces no duplicate identifiers

```
lemma distinct-upto:
  shows i ≤ MAX-STR ⟹ distinct (fin-string-upto i)
proof (induction i)
  case 0
  then show ?case by(auto)
next
  case (Suc j) then
  have jLen:Suc j ≤ MAX-STR
    and IH:distinct (fin-string-upto j) by auto
  have distinct-char:distinct (enum-class.enum :: char list)
    by (auto simp add: distinct-map enum-char-unfold)
 have diseq:⋀ x y. y ∈ set (fin-string-upto j) ⟹ fin-string-empty ≠ fin-string-cons
x y
     using Rep-fin-string-inverse jLen apply(auto simp add: fin-string-empty-def
fin-string-cons-def)
   using fin-string-empty.rep-eq le-zero-eq list.size not-less-eq-eq zero-le Abs-fin-string-inject
    by (metis,auto)
  show ?case
    apply(auto simp add: Let-def)
    subgoal for x xa using diseq by auto
    apply(rule distinct-concat)
    subgoal
      apply(auto simp add: distinct-map)
       apply(rule distinct-char)
      apply(rule inj-on-subset)
       apply(rule injI)
       apply(auto simp add: fin-string-cons-def)
    proof −
      fix x y
      let ?l = (λxa x. Abs-fin-string
        (if MAX-STR ≤ length (Rep-fin-string xa)
         then Rep-fin-string xa
         else x # Rep-fin-string xa))
      assume a1:∀ xa∈set (fin-string-upto j). ?l xa x = ?l xa y
      then have a2:⋀xa. (List.member (fin-string-upto j) xa) ⟹ ?l xa x = ?l xa
y
        by force
      then have Abs-fin-string [x] = Abs-fin-string [y] ∨ (MAX-STR::nat) = 0
        using a2 fin-string-empty.rep-eq fin-string-nil by force
      then show x = y
           by (metis Abs-fin-string-inverse jLen le-zero-eq length-Cons list.inject
list.size(3)
           mem-Collect-eq nat.distinct(1) not-less-eq-eq)
    qed
    subgoal for ys
      apply(auto simp add: fin-string-cons-def)
    proof −
      fix c :: char
      assume c:c ∈ set enum-class.enum
```

```
      assume ys:ys=map(map-fun Rep-fin-string Abs-fin-string (string-cons c))
(fin-string-upto j)
    showdistinct(map(map-fun Rep-fin-string Abs-fin-string (string-cons c)) (fin-string-upto
j))
      unfolding distinct-map apply(rule conjI)
       apply(rule IH)
      apply(rule inj-onI)
      apply(auto)
      subgoal for x y
        using jLen fin-string-upto-length[of j x] fin-string-upto-length[of j y]
        unfolding ilength-def apply auto
     by (metis (mono-tags, opaque-lifting) Rep-fin-string-inverse fin-string-cons.rep-eq
le-trans
            list.inject not-less-eq-eq string-cons.simps)
        done
    qed
    apply (auto simp add: fin-string-cons-def enum-UNIV not-le split: if-splits)
    using fin-string-upto-length ilength.rep-eq jLen apply fastforce
    using fin-string-upto-length ilength.rep-eq jLen apply fastforce
    using fin-string-upto-length ilength.rep-eq jLen apply fastforce
      apply (metis Abs-fin-string-inverse Suc-le-eq length-Cons mem-Collect-eq
nth-Cons-0)
    using fin-string-upto-length ilength.rep-eq jLen apply fastforce
    using fin-string-upto-length ilength.rep-eq jLen apply fastforce
    using fin-string-upto-length ilength.rep-eq jLen apply fastforce
      apply (metis Abs-fin-string-inverse Suc-le-eq length-Cons mem-Collect-eq
nth-Cons-0)
    done
  qed
```

Finite strings are an enumeration type

```
instantiation fin-string :: enum begin
definition enum-fin-string
  where enum-fin-string-def[code]:enum-fin-string ≡ fin-string-enum
definition enum-all-fin-string
  where enum-all-fin-string[code]:enum-all-fin-string ≡ fin-string-enum-all
definition enum-ex-fin-string
  where enum-ex-fin-string[code]:enum-ex-fin-string ≡ fin-string-enum-ex
lemma enum-ALL: (UNIV::fin-string set) = set enum-class.enum
  apply (auto simp add:enum-fin-string-def fin-string-enum-def vals-def)
 apply (metis Abs-fin-string-cases List.member-iff fin-string-length mem-Collect-eq)
  done

lemma vals-ALL:set (vals::fin-string list) = UNIV
  apply (auto simp add: vals-def)
  subgoal for s
    using fin-string-length [of ‹Rep-fin-string s› MAX-STR] Rep-fin-string [of s]
    apply (simp add: Rep-fin-string-inverse)
    done
```

done

lemma *setA*:
assumes *set*:$\bigwedge y.\ y \in set\ L \Longrightarrow P\ y$
shows *list-all P L*
using *set* **by** (*simp add*: *list.pred-set*)

lemma *setE*:
assumes *set*: $y \in set\ L$
assumes *P*:*P y*
shows *list-ex P L*
using *set P list-ex-iff* **by** *auto*

instance
apply(*standard*)
apply(*rule enum-ALL*)
by (*auto simp add*: *fin-string-enum-all-def list-all-iff vals-ALL setA setE enum-all-fin-string enum-ALL fin-string-enum-def vals-def enum-fin-string-def distinct-upto list-ex-iff*

enum-ex-fin-string fin-string-enum-ex-def)
end

instantiation *fin-string* :: *equal* **begin**
definition *equal-fin-string* :: *fin-string* ⇒ *fin-string* ⇒ *bool*
where [*code*]:*equal-fin-string X Y* = ($X \leq Y \wedge Y \leq X$)
instance
apply(*standard*)
by(*auto simp add*: *equal-fin-string-def*)
end
end

Interpreter.thy defines a simple programming language over interval-valued variables and executable semantics (interpreter) for that language. We then prove that the interpretation of interval terms is a sound over-approximation of a real-valued semantics of the same language.

Our language is a version of first order dynamic logic-style regular programs. We use a finite identifier space for compatibility with Differential-Dynamic-Logic, where identifier finiteness is required to treat program states as Banach spaces to enable differentiation.

theory *Interpreter*
imports
Complex-Main
Finite-String
Interval-Word32
begin

5 Syntax

Our term language supports variables, polynomial arithmetic, and extrema. This choice was made based on the needs of the original paper and could be extended if necessary.

datatype *trm* =
  Var fin-string
| *Const lit*
| *Plus trm trm*
| *Times trm trm*
| *Neg trm*
| *Max trm trm*
| *Min trm trm*
| *Abs trm*

Our statement language is nondeterministic first-order regular programs. This coincides with the discrete subset of hybrid programs from the dL entry.

Our assertion language are the formulas of first-order dynamic logic

datatype *prog* =
  Assign fin-string trm (**infixr** ‹:=› *10*)
| *AssignAny fin-string*
| *Test formula* (‹*?*›)
| *Choice prog prog* (**infixl** ‹∪∪› *10*)
| *Sequence prog prog* (**infixr** ‹;;› *8*)
| *Loop prog* (‹-**›)

and *formula* =
  Geq trm trm
| *Not formula* (‹!›)
| *And formula formula* (**infixl** ‹&&› *8*)
| *Exists fin-string formula*
| *Diamond prog formula* (‹(⟨ - ⟩ -)› *10*)

Derived forms

definition *Or* :: *formula* ⇒ *formula* ⇒ *formula* (**infixl** ‹||› *7*)
where *or-simp*[*simp*]:*Or P Q* = *Not* (*And* (*Not P*) (*Not Q*))

definition *Equals* :: *trm* ⇒ *trm* ⇒ *formula*
where *equals-simp*[*simp*]:*Equals ϑ ϑ′* = (*And* (*Geq ϑ ϑ′*) (*Geq ϑ′ ϑ*))

definition *Greater* :: *trm* ⇒ *trm* ⇒ *formula*
where *greater-simp*[*simp*]:*Greater ϑ ϑ′* = *Not* (*Geq ϑ′ ϑ*)

definition *Leq* :: *trm* ⇒ *trm* ⇒ *formula*
where *leq-simp*[*simp*]:*Leq ϑ ϑ′* = (*Geq ϑ′ ϑ*)

definition *Less* :: *trm* ⇒ *trm* ⇒ *formula*

where *less-simp*[*simp*]:*Less* $\vartheta\ \vartheta' = (Not\ (Geq\ \vartheta\ \vartheta'))$

6 Semantics

States over reals vs. word intervals which contain them

type-synonym *rstate* = *fin-string* ⇒ *real*
type-synonym *wstate* = (*fin-string* + *fin-string*) ⇒ *word*

definition *wstate*::*wstate* ⇒ *prop*
where *wstate-def*[*simp*]:*wstate* $\nu \equiv (\bigwedge i.\ word\ (\nu\ (Inl\ i)) \wedge word\ (\nu\ (Inr\ i)))$

Interpretation of a term in a state

inductive *rtsem* :: *trm* ⇒ *rstate* ⇒ *real* ⇒ *bool* (‹([-]- ↓ -)› *10*)
where
rtsem-Const:*Rep-bword w* $\equiv_E$ *r* ⟹ ([*Const w*]ν ↓ *r*)
| *rtsem-Var*:([*Var x*]ν ↓ ν *x*)
| *rtsem-Plus*:⟦([ϑ_1]ν ↓ *r1*); ([ϑ_2]ν ↓ *r2*)⟧ ⟹ ([*Plus* ϑ_1 ϑ_2]ν ↓ (*r1* + *r2*))
| *rtsem-Times*:⟦([ϑ_1]ν ↓ *r1*); ([ϑ_2]ν ↓ *r2*)⟧ ⟹ ([*Times* ϑ_1 ϑ_2]ν ↓ (*r1* ∗ *r2*))
| *rtsem-Max*:⟦([ϑ_1]ν ↓ *r1*); ([ϑ_2]ν ↓ *r2*)⟧ ⟹ ([*Max* ϑ_1 ϑ_2]ν ↓ (*max r1 r2*))
| *rtsem-Min*:⟦([ϑ_1]ν ↓ *r1*); ([ϑ_2]ν ↓ *r2*)⟧ ⟹ ([*Min* ϑ_1 ϑ_2]ν ↓ (*min r1 r2*))
| *rtsem-Abs*:⟦([ϑ_1]ν ↓ *r1*)⟧ ⟹ ([*Abs* ϑ_1]ν ↓ (*abs r1*))
| *rtsem-Neg*:([ϑ]ν ↓ *r*) ⟹ ([*Neg* ϑ]ν ↓ −*r*)

inductive-simps
rtsem-Const-simps[*simp*] : ([(*Const w*)]ν ↓ *r*)
and *rtsem-Var-simps*[*simp*] : ([*Var x*]ν ↓ *r*)
and *rtsem-PlusU-simps*[*simp*] : ([*Plus* ϑ_1 ϑ_2]ν ↓ *r*)
and *rtsem-TimesU-simps*[*simp*] : ([*Times* ϑ_1 ϑ_2]ν ↓ *r*)
and *rtsem-Max-simps*[*simp*] : ([*Max* ϑ_1 ϑ_2] ν ↓ *r*)
and *rtsem-Min-simps*[*simp*] : ([*Min* ϑ_1 ϑ_2] ν ↓ *r*)
and *rtsem-Abs-simps*[*simp*] : ([*Abs* ϑ] ν ↓ *r*)
and *rtsem-Neg-simps*[*simp*] : ([*Neg* ϑ] ν ↓ *r*)

definition *set-less* :: *real set* ⇒ *real set* ⇒ *bool* (**infix** ‹$<_S$› *10*)
where *set-less A B* ≡ (∀ *x y*. *x* ∈ *A* ∧ *y* ∈ *B* ⟶ *x* < *y*)

definition *set-geq* :: *real set* ⇒ *real set* ⇒ *bool* (**infix** ‹$\geq_S$› *10*)
where *set-geq A B* ≡ (∀ *x y*. *x* ∈ *A* ∧ *y* ∈ *B* ⟶ *x* ≥ *y*)

Interpretation of an assertion in a state

inductive *rfsem* :: *formula* ⇒ *rstate* ⇒ *bool* ⇒ *bool* (‹([-]-) ⌊ -› *20*)
where
rGreaterT:⟦([ϑ_1]ν ↓ *r1*); ([ϑ_2]ν ↓ *r2*)⟧ ⟹ *r1* > *r2* ⟹ ([*Greater* ϑ_1 ϑ_2] ν ⌊ *True*)
| *rGreaterF*:⟦([ϑ_1]ν ↓ *r1*); ([ϑ_2]ν ↓ *r2*)⟧ ⟹ *r2* ≥ *r1* ⟹ ([*Greater* ϑ_1 ϑ_2] ν ⌊ *False*)
| *rGeqT*:⟦([ϑ_1]ν ↓ *r1*); ([ϑ_2]ν ↓ *r2*)⟧ ⟹ *r1* ≥ *r2* ⟹ ([*Geq* ϑ_1 ϑ_2] ν ⌊ *True*)
| *rGeqF*:⟦([ϑ_1]ν ↓ *r1*); ([ϑ_2]ν ↓ *r2*)⟧ ⟹ *r2* > *r1* ⟹ ([*Geq* ϑ_1 ϑ_2] ν ⌊ *False*)

| *rEqualsT*:⟦([ϑ_1]ν ↓ *r1*); ([ϑ_2]ν ↓ *r2*)⟧ ⟹ *r1* = *r2* ⟹ ([*Equals* ϑ_1 ϑ_2] ν ⌊ *True*)
| *rEqualsF*:⟦([ϑ_1]ν ↓ *r1*); ([ϑ_2]ν ↓ *r2*)⟧ ⟹ *r1* ≠ *r2* ⟹ ([*Equals* ϑ_1 ϑ_2] ν ⌊ *False*)
| *rAndT*:⟦([φ]ν ⌊ *True*); ([ψ]ν ⌊ *True*)⟧ ⟹ ([*And* φ ψ]ν ⌊ *True*)
| *rAndF1*:([φ]ν ⌊ *False*) ⟹ ([*And* φ ψ]ν ⌊ *False*)
| *rAndF2*:([ψ]ν ⌊ *False*) ⟹ ([*And* φ ψ]ν ⌊ *False*)
| *rOrT1*:([φ]ν ⌊ *True*) ⟹ ([*Or* φ ψ]ν ⌊ *True*)
| *rOrT2*:([ψ]ν ⌊ *True*) ⟹ ([*Or* φ ψ]ν ⌊ *True*)
| *rOrF*:⟦([φ]ν ⌊ *False*) ;([ψ]ν ⌊ *False*)⟧ ⟹ ([*And* φ ψ]ν ⌊ *False*)
| *rNotT*:([φ]ν ⌊ *False*) ⟹ ([(*Not* φ)]ν ⌊ *True*)
| *rNotF*:([φ]ν ⌊ *True*) ⟹ ([(*Not* φ)]ν ⌊ *False*)

inductive-simps
  rfsem-Greater-simps[*simp*]: ([*Greater* ϑ_1 ϑ_2]ν ⌊ *b*)
and *rfsem-Geq-simps*[*simp*]: ([*Geq* ϑ_1 ϑ_2]ν ⌊ *b*)
and *rfsem-Equals-simps*[*simp*]: ([*Equals* ϑ_1 ϑ_2]ν ⌊ *b*)
and *rfsem-And-simps*[*simp*]: ([*And* φ ψ]ν ⌊ *b*)
and *rfsem-Or-simps*[*simp*]: ([(*Or* φ ψ)]ν ⌊ *b*)
and *rfsem-Not-simps*[*simp*]: ([*Not* φ]ν ⌊ *b*)

Interpretation of a program is a transition relation on states

inductive *rpsem* :: *prog* ⇒ *rstate* ⇒ *rstate* ⇒ *bool* (‹([-]-) ⌊ -› *20*)
where
  rTest[*simp*]:⟦([φ]ν ⌊ *True*); ν = ω⟧ ⟹ ([? φ]ν ⌊ ω)
| *rSeq*[*simp*]:⟦([α]ν ⌊ μ); ([β]μ ⌊ ω)⟧ ⟹ ([α;; β]ν ⌊ ω)
| *rAssign*[*simp*]:⟦([ϑ]ν ↓ *r*); ω = (ν (*x* := *r*))⟧ ⟹ ([*Assign x* ϑ]ν ⌊ ω)
| *rChoice1*[*simp*]:([α]ν ⌊ ω) ⟹ ([*Choice* α β]ν ⌊ ω)
| *rChoice2*[*simp*]:([β]ν ⌊ ω) ⟹ ([*Choice* α β]ν ⌊ ω)

inductive-simps
  rpsem-Test-simps[*simp*]: ([? φ]ν ⌊ *b*)
and *rpsem-Seq-simps*[*simp*]: ([α;; β]ν ⌊ *b*)
and *rpsem-Assign-simps*[*simp*]: ([*Assign x* ϑ]ν ⌊ *b*)
and *rpsem-Choice-simps*[*simp*]: ([*Choice* α β]ν ⌊ *b*)

Upper bound of arbitrary term

fun *wtsemU* :: *trm* ⇒ *wstate* ⇒ *word* * *word* (‹([-]<>-)› *20*)
where ([*Const r*]<>ν) = (*Rep-bword r*::*word*, *Rep-bword r*)
| *wVarU*:([*Var x*]<>ν) = (ν (*Inl x*), ν (*Inr x*))
| *wPlusU*:([*Plus* ϑ_1 ϑ_2]<> ν) =
  (*let* (*l1*, *u1*) = [ϑ_1]<> ν *in*
   let (*l2*, *u2*) = [ϑ_2]<> ν *in*
   (*pl l1 l2*, *pu u1 u2*))
| *wTimesU*:([(*Times* ϑ_1 ϑ_2)]<> ν) =
  (*let* (*l1*, *u1*) = [ϑ_1]<> ν *in*
   let (*l2*, *u2*) = [ϑ_2]<> ν *in*
   (*tl l1 u1 l2 u2*, *tu l1 u1 l2 u2*))
| *wMaxU*:([(*Max* ϑ_1 ϑ_2)]<> ν) =
  (*let* (*l1*, *u1*) = [ϑ_1]<> ν *in*
   let (*l2*, *u2*) = [ϑ_2]<> ν *in*

$(wmax\ l1\ l2,\ wmax\ u1\ u2))$
| $wMinU$:$([(Min\ \vartheta_1\ \vartheta_2)]<>\ \nu) =$
$(let\ (l1,\ u1) = [\vartheta_1]<>\ \nu\ in$
$let\ (l2,\ u2) = [\vartheta_2]<>\ \nu\ in$
$(wmin\ l1\ l2,\ wmin\ u1\ u2))$
| $wNegU$:$([(Neg\ \vartheta)]<>\ \nu) =$
$(let\ (l,\ u) = [\vartheta]<>\ \nu\ in$
$(wneg\ u,\ wneg\ l))$
| $wAbsU$:$([(Abs\ \vartheta_1)]<>\ \nu) =$
$(let\ (l1,\ u1) = [\vartheta_1]<>\ \nu\ in$
$(wmax\ l1\ (wneg\ u1),\ wmax\ u1\ (wneg\ l1)))$

inductive $wfsem :: formula \Rightarrow wstate \Rightarrow bool \Rightarrow bool\ (\langle([[-]]- \downarrow -)\rangle\ 20)$
where
$wGreaterT$:$wgreater\ (fst\ ([\vartheta_1]<>\nu))\ (snd\ ([\vartheta_2]<>\nu)) \Longrightarrow\ ([[(Greater\ \vartheta_1\ \vartheta_2)]]\nu \downarrow True)$
| $wGreaterF$:$wgeq\ (fst\ ([\vartheta_2]<>\nu))\ (snd\ ([\vartheta_1]<>\nu)) \Longrightarrow\ ([[(Greater\ \vartheta_1\ \vartheta_2)]]\nu \downarrow False)$
| $wGeqT$:$wgeq\ (fst\ ([\vartheta_1]<>\ \nu))\ (snd\ ([\vartheta_2]<>\nu)) \Longrightarrow\ ([[(Geq\ \vartheta_1\ \vartheta_2)]]\nu \downarrow True)$
| $wGeqF$:$wgreater\ (fst\ ([\vartheta_2]<>\nu))\ (snd\ ([\vartheta_1]<>\nu)) \Longrightarrow\ ([[(Geq\ \vartheta_1\ \vartheta_2)]]\nu \downarrow False)$
| $wEqualsT$:$[\![(fst\ ([\vartheta_2]<>\nu) = snd\ ([\vartheta_2]<>\nu));\ (snd\ ([\vartheta_2]<>\nu) = snd\ ([\vartheta_1]<>\nu));$

$(snd\ ([\vartheta_1]<>\nu) = fst\ ([\vartheta_1]<>\nu));\ (fst\ ([\vartheta_2]<>\nu) \neq NEG\text{-}INF);$
$(fst\ ([\vartheta_2]<>\nu) \neq POS\text{-}INF)]\!]$
$\Longrightarrow ([[Equals\ \vartheta_1\ \vartheta_2]]\ \nu \downarrow True)$
| $wEqualsF1$:$wgreater\ (fst\ ([\vartheta_1]<>\ \nu))\ (snd\ ([\vartheta_2]<>\nu)) \Longrightarrow\ ([[Equals\ \vartheta_1\ \vartheta_2]]\ \nu \downarrow False)$
| $wEqualsF2$:$wgreater\ (fst\ ([\vartheta_2]<>\ \nu))\ (snd\ ([\vartheta_1]<>\nu)) \Longrightarrow\ ([[Equals\ \vartheta_1\ \vartheta_2]]\ \nu \downarrow False)$
| $wAndT$:$[\![[[\varphi]]\nu \downarrow True;\ [[\psi]]\nu \downarrow True]\!] \Longrightarrow ([[And\ \varphi\ \psi]]\nu \downarrow True)$
| $wAndF1$:$[[\varphi]]\nu \downarrow False \Longrightarrow ([[And\ \varphi\ \psi]]\nu \downarrow False)$
| $wAndF2$:$[[\psi]]\nu \downarrow False \Longrightarrow ([[And\ \varphi\ \psi]]\nu \downarrow False)$
| $wOrT1$:$([[\varphi]]\nu \downarrow True) \Longrightarrow ([[Or\ \varphi\ \psi]]\nu \downarrow True)$
| $wOrT2$:$([[\psi]]\nu \downarrow True) \Longrightarrow ([[Or\ \varphi\ \psi]]\nu \downarrow True)$
| $wOrF$:$[\![[[\varphi]]\nu \downarrow False;\ [[\psi]]\nu \downarrow False]\!] \Longrightarrow ([[And\ \varphi\ \psi]]\nu \downarrow False)$
| $wNotT$:$([[\varphi]]\nu \downarrow False) \Longrightarrow ([[Not\ \varphi]]\nu \downarrow True)$
| $wNotF$:$([[\varphi]]\nu \downarrow True) \Longrightarrow ([[Not\ \varphi]]\nu \downarrow False)$

inductive-simps
$wfsem\text{-}Gr\text{-}simps[simp]$: $([[Le\ \vartheta_1\ \vartheta_2]]\nu \downarrow b)$
and $wfsem\text{-}And\text{-}simps[simp]$: $([[And\ \varphi\ \psi]]\nu \downarrow b)$
and $wfsem\text{-}Or\text{-}simps[simp]$: $([[Or\ \varphi\ \psi]]\nu \downarrow b)$
and $wfsem\text{-}Not\text{-}simps[simp]$: $([[Not\ \varphi]]\nu \downarrow b)$
and $wfsem\text{-}Equals\text{-}simps[simp]$: $([[Equals\ \vartheta_1\ \vartheta_2]]\nu \downarrow b)$

Program semantics

inductive $wpsem :: prog \Rightarrow\ wstate \Rightarrow wstate \Rightarrow bool\ (\langle([[-]]- \downarrow -)\rangle\ 20)$
where
$wTest$:$([[\varphi]]\nu \downarrow True) \Longrightarrow \nu = \omega \Longrightarrow ([[?\ \varphi]]\ \nu \downarrow \omega)$

| *wSeq*:([[α]]$\nu \downarrow \mu$) $\Longrightarrow$ ([[β]] $\mu \downarrow \omega$) $\Longrightarrow$ ([[α;; β]] $\nu \downarrow \omega$)
| *wAssign*:ω = ((ν ((*Inr x*) := *snd*([ϑ]<>ν))) ((*Inl x*) := *fst*([ϑ]<>ν))) $\Longrightarrow$ ([[*Assign* *x* ϑ]] $\nu \downarrow \omega$)
| *wChoice1*[*simp*]:([[α]]$\nu \downarrow \omega$) $\Longrightarrow$ ([[*Choice* α β]]$\nu \downarrow \omega$)
| *wChoice2*[*simp*]:([[β]]$\nu \downarrow \omega$) $\Longrightarrow$ ([[*Choice* α β]]$\nu \downarrow \omega$)

inductive-simps
wpsem-Test-simps[*simp*]: ([[*Test* φ]]$\nu \downarrow b$)
and *wpsem-Seq-simps*[*simp*]: ([[α;; β]]$\nu \downarrow b$)
and *wpsem-Assign-simps*[*simp*]: ([[*Assign* *x* ϑ]]$\nu \downarrow b$)
and *wpsem-Choice-simps*[*simp*]: ([[*Choice* α β]]$\nu \downarrow b$)

lemmas *real-max-mono* = *Lattices.linorder-class.max.mono*
lemmas *real-minus-le-minus* = *Groups.ordered-ab-group-add-class.neg-le-iff-le*

Interval state consists of upper and lower bounds for each real variable

inductive *represents-state*::*wstate* $\Rightarrow$ *rstate* $\Rightarrow$ *bool* (**infix** ‹*REP*› *10*)
where *REPI*:($\bigwedge$*x*. (ν (*Inl x*) $\equiv_L$ ν' *x*) $\wedge$ (ν (*Inr x*) $\equiv_U$ ν' *x*)) $\Longrightarrow$ (ν *REP* ν')

inductive-simps *repstate-simps*:ν *REP* ν'

7 Soundness proofs

Interval term valuation soundly contains real valuation

lemma *trm-sound*:
fixes ϑ::*trm*
shows ([ϑ]$\nu' \downarrow r$) $\Longrightarrow$ (ν *REP* ν') $\Longrightarrow$ ([ϑ]<>ν) $\equiv_P$ *r*
proof (*induction rule*: *rtsem.induct*)
case *rtsem-Const*
fix *w r* ν'
show *Rep-bword w* $\equiv_E$ *r* $\Longrightarrow$ ν *REP* ν' $\Longrightarrow$ [*Const w*]<>ν $\equiv_P$ *r*
using *repU-def repL-def repP-def repe.simps rep-simps repstate-simps*
by *auto*
next
case *rtsem-Var*
fix *x* ν'
show ν *REP* ν' $\Longrightarrow$ [*Var x*]<>ν $\equiv_P$ ν' *x*
by(*auto simp add*: *repU-def repL-def repP-def repe.simps rep-simps repstate-simps*)
next
case *rtsem-Plus*
fix ϑ_1 :: *trm* **and** ν':: *rstate* **and** *r1* **and** ϑ_2 :: *trm* **and** *r2*
assume *rep*:ν *REP* ν'
assume *eval1*:[ϑ_1]$\nu' \downarrow$ *r1*
assume (ν *REP* ν' $\Longrightarrow$ [ϑ_1]<>ν $\equiv_P$ *r1*)
then have *IH1*:[ϑ_1]<>ν $\equiv_P$ *r1* **using** *rep* **by** *auto*
assume *eval2*:[ϑ_2]$\nu' \downarrow$ *r2*
assume (ν *REP* ν' $\Longrightarrow$ [ϑ_2]<>ν $\equiv_P$ *r2*)
then have *IH2*:[ϑ_2]<>ν $\equiv_P$ *r2* **using** *rep* **by** *auto*

```
  obtain l1 u1 l2 u2 where
      lu1:(l1, u1) = ([ϑ1]<> ν)
   and lu2:(l2, u2) = ([ϑ2]<> ν)
   using IH1 IH2 repP-def by auto
  from lu1 and lu2 have
      lu1':([ϑ1]<> ν) = (l1, u1)
   and lu2':([ϑ2]<> ν) = (l2, u2)
   by auto
  have l1:l1 ≡L r1 using IH1 lu1 unfolding repP-def by auto
  have u1:u1 ≡U r1 using IH1 lu1 unfolding repP-def by auto
  have l2:l2 ≡L r2 using IH2 lu2 unfolding repP-def by auto
  have u2:u2 ≡U r2 using IH2 lu2 unfolding repP-def by auto
  then have ([(Plus ϑ1 ϑ2)]<>ν) = (pl l1 l2, pu u1 u2)
   using lu1' lu2' by auto
  have lBound:(pl l1 l2 ≡L r1 + r2)
    using l1 l2 pl-lemma by auto
  have uBound:(pu u1 u2 ≡U r1 + r2)
    using pu-lemma[OF u1 u2] by auto
  have (pl l1 l2, pu u1 u2) ≡P (r1 + r2)
    unfolding repP-def Let-def using lBound uBound by auto
  then show[Plus ϑ1 ϑ2]<>ν ≡P r1 + r2
    using lu1' lu2' by auto
next
 case rtsem-Times
   fix ϑ1 :: trm and ν' r1 and ϑ2 :: trm and r2
   assume eval1:[ϑ1]ν' ↓ r1
   assume eval2:[ϑ2]ν' ↓ r2
   assume rep:ν REP ν'
   assume (ν REP ν' ⟹ ([ϑ1]<>ν ≡P r1))
   then have IH1:[ϑ1]<>ν ≡P r1 using rep by auto
   assume (ν REP ν' ⟹ ([ϑ2]<>ν ≡P r2))
   then have IH2:[ϑ2]<>ν ≡P r2 using rep by auto
    obtain l1 u1 l2 u2 where
      lu1:([ϑ1]<> ν) = (l1, u1)
    and lu2:([ϑ2]<> ν) = (l2, u2)
    using IH1 IH2 repP-def by auto
  have l1:l1 ≡L r1 using IH1 lu1 unfolding repP-def by auto
  have u1:u1 ≡U r1 using IH1 lu1 unfolding repP-def by auto
  have l2:l2 ≡L r2 using IH2 lu2 unfolding repP-def by auto
  have u2:u2 ≡U r2 using IH2 lu2 unfolding repP-def by auto
  then have ([(Times ϑ1  ϑ2)]<>ν) = (tl l1 u1 l2 u2, tu l1 u1 l2 u2)
   using lu1 lu2 unfolding wTimesU Let-def by auto
  have lBound:(tl l1 u1 l2 u2 ≡L r1 ∗ r2)
    using l1 u1 l2 u2 tl-lemma by auto
  have uBound:(tu l1 u1 l2 u2 ≡U r1 ∗ r2)
    using l1 u1 l2 u2 tu-lemma by auto
  have (tl l1 u1 l2 u2, tu l1 u1 l2 u2) ≡P (r1 ∗ r2)
    unfolding repP-def Let-def using lBound uBound by auto
  then show [Times ϑ1 ϑ2]<>ν ≡P r1 ∗ r2
```

using *lu1 lu2* **by** *auto*
next
case *rtsem-Max*
fix ϑ_1 :: *trm* **and** ν' *r1* **and** ϑ_2 :: *trm* **and** *r2*
assume *eval1*:$([\vartheta_1]\nu' \downarrow r1)$
assume *eval2*:$([\vartheta_2]\nu' \downarrow r2)$
assume *rep*:ν *REP* ν'
assume $(\nu\ REP\ \nu' \Longrightarrow [\vartheta_1]<>\nu \equiv_P r1)$
then have *IH1*:$[\vartheta_1]<>\nu \equiv_P r1$ **using** *rep* **by** *auto*
assume $(\nu\ REP\ \nu' \Longrightarrow [\vartheta_2]<>\nu \equiv_P r2)$
then have *IH2*:$[\vartheta_2]<>\nu \equiv_P r2$ **using** *rep* **by** *auto*
obtain *l1 u1 l2 u2* **where**
$lu1$:$([\vartheta_1]<>\ \nu) = (l1,\ u1)$
and *lu2*:$([\vartheta_2]<>\ \nu) = (l2,\ u2)$
using *IH1 IH2 repP-def* **by** *auto*
from *IH1 IH2*
obtain *ub1 ub2 lb1 lb2*:: *real*
where *urep1*:$(ub1 \geq r1) \wedge (snd\ ([\vartheta_1]<>\nu) \equiv_E ub1)$
and *urep2*:$(ub2 \geq r2) \wedge (snd\ ([\vartheta_2]<>\nu) \equiv_E ub2)$
and *lrep1*:$(lb1 \leq r1) \wedge (fst\ ([\vartheta_1]<>\nu) \equiv_E lb1)$
and *lrep2*:$(lb2 \leq r2) \wedge (fst\ ([\vartheta_2]<>\nu) \equiv_E lb2)$
using *repP-def repU-def repL-def* **by** *auto*
have *lbound*:$wmax\ l1\ l2 \equiv_L max\ r1\ r2$
by (*metis dual-order.trans fst-conv le-cases lrep1 lrep2 lu1 lu2 max-def repL-def wmax.elims*)
have *ubound*:$wmax\ u1\ u2 \equiv_U max\ r1\ r2$
by (*metis real-max-mono lu1 lu2 repU-def snd-conv urep1 urep2 wmax-lemma*)
have $([trm.Max\ \vartheta_1\ \vartheta_2]<>\nu) = (wmax\ l1\ l2,\ wmax\ u1\ u2)$
using *lu1 lu2* **unfolding** *wMaxU Let-def*
by (*simp*)
then show $[trm.Max\ \vartheta_1\ \vartheta_2]<>\nu \equiv_P max\ r1\ r2$
unfolding *repP-def*
using *lbound ubound lu1 lu2* **by** *auto*
next
case *rtsem-Min*
fix ϑ_1 :: *trm* **and** ν' *r1* **and** ϑ_2 :: *trm* **and** *r2*
assume *eval1*:$([\vartheta_1]\nu' \downarrow r1)$
assume *eval2*:$([\vartheta_2]\nu' \downarrow r2)$
assume *rep*:ν *REP* ν'
assume $(\nu\ REP\ \nu' \Longrightarrow [\vartheta_1]<>\nu \equiv_P r1)$
then have *IH1*:$[\vartheta_1]<>\nu \equiv_P r1$ **using** *rep* **by** *auto*
assume $(\nu\ REP\ \nu' \Longrightarrow [\vartheta_2]<>\nu \equiv_P r2)$
then have *IH2*:$[\vartheta_2]<>\nu \equiv_P r2$ **using** *rep* **by** *auto*
obtain *l1 u1 l2 u2* **where**
$lu1$:$([\vartheta_1]<>\ \nu) = (l1,\ u1)$
and *lu2*:$([\vartheta_2]<>\ \nu) = (l2,\ u2)$
using *IH1 IH2 repP-def* **by** *auto*
from *IH1 IH2*
obtain *ub1 ub2 lb1 lb2*:: *real*

where *urep1*:$(ub1 \geq r1) \wedge (snd\ ([\vartheta_1]{<>}\nu) \equiv_E ub1)$
and *urep2*:$(ub2 \geq r2) \wedge (snd\ ([\vartheta_2]{<>}\nu) \equiv_E ub2)$
and *lrep1*:$(lb1 \leq r1) \wedge (fst\ ([\vartheta_1]{<>}\nu) \equiv_E lb1)$
and *lrep2*:$(lb2 \leq r2) \wedge (fst\ ([\vartheta_2]{<>}\nu) \equiv_E lb2)$
using *prod.case-eq-if repP-def repU-def repL-def* **by** *auto*
have *lbound*:$wmin\ l1\ l2 \equiv_L min\ r1\ r2$
by (*metis fst-conv lrep1 lrep2 lu1 lu2 min.mono repL-def wmin-lemma*)
have *ubound*:$wmin\ u1\ u2 \equiv_U min\ r1\ r2$
using *lu1 lu2 min-le-iff-disj repU-def urep1 urep2* **by** *auto*
have $([Min\ \vartheta_1\ \vartheta_2]{<>}\nu) = (wmin\ l1\ l2,\ wmin\ u1\ u2)$
using *lu1 lu2* **unfolding** *wMinU Let-def* **by** *auto*
then show $[Min\ \vartheta_1\ \vartheta_2]{<>}\nu \equiv_P min\ r1\ r2$
unfolding *repP-def*
using *lbound ubound lu1 lu2* **by** *auto*
next
case *rtsem-Neg*
fix ϑ :: *trm* **and** $\nu'\ r$
assume *eval*:$[\vartheta]\nu' \downarrow r$
assume *rep*:$\nu\ REP\ \nu'$
assume $(\nu\ REP\ \nu' \Longrightarrow [\vartheta]{<>}\nu \equiv_P r)$
then have *IH*:$[\vartheta]{<>}\nu \equiv_P r$ **using** *rep* **by** *auto*
obtain *l1 u1* **where**
lu:$([\vartheta]{<>}\ \nu) = (l1,\ u1)$
using *IH repP-def* **by** *auto*
from *IH*
obtain *ub lb*:: *real*
where *urep*:$(ub \geq r) \wedge (snd\ ([\vartheta]{<>}\nu) \equiv_E ub)$
and *lrep*:$(lb \leq r) \wedge (fst\ ([\vartheta]{<>}\nu) \equiv_E lb)$
using *repP-def repU-def repL-def* **by** *auto*
have *ubound*:$((wneg\ u1) \equiv_L (uminus\ r))$
by (*metis real-minus-le-minus lu repL-def snd-conv urep wneg-lemma*)
have *lbound*:$((wneg\ l1) \equiv_U (uminus\ r))$
using *real-minus-le-minus lu repU-def lrep wneg-lemma*
by (*metis fst-conv*)
show $[Neg\ \vartheta]{<>}\nu \equiv_P -\ r$
unfolding *repP-def Let-def* **using** *ubound lbound lu*
by (*auto*)
next
case *rtsem-Abs*
fix ϑ :: *trm* **and** $\nu'\ r$
assume *eval*:$[\vartheta]\nu' \downarrow r$
assume *rep*:$\nu\ REP\ \nu'$
assume $(\nu\ REP\ \nu' \Longrightarrow [\vartheta]{<>}\nu \equiv_P r)$
then have *IH*:$[\vartheta]{<>}\nu \equiv_P r$ **using** *rep* **by** *auto*
obtain *l1 u1* **where**
lu:$([\vartheta]{<>}\ \nu) = (l1,\ u1)$
using *IH repP-def* **by** *auto*
from *IH*
obtain *ub lb*:: *real*

where *urep*:$(ub \geq r) \wedge (snd\ ([\vartheta]<>\nu) \equiv_E ub)$
and *lrep*:$(lb \leq r) \wedge (fst\ ([\vartheta]<>\nu) \equiv_E lb)$
using *prod.case-eq-if repP-def repU-def repL-def* **by** *auto*
have *lbound*:$wmax\ l1\ (wneg\ u1) \equiv_L (abs\ r)$
apply(*simp only*: *repL-def*)
apply(*rule exI*[**where** x=*max lb* (− *ub*)])
apply(*rule conjI*)
using *lrep wmax-lemma lu urep wneg-lemma* **by** *auto*
have *ubound*:$(wmax\ u1\ (wneg\ l1) \equiv_U (abs\ r))$
apply(*simp only*: *repU-def*)
apply(*rule exI*[**where** x=*max ub* (− *lb*)])
using *lrep wmax-lemma lu urep wneg-lemma* **by** *auto*
show $[Abs\ \vartheta]<>\nu \equiv_P abs\ r$
using *repP-def Let-def ubound lbound lu lu wAbsU* **by** *auto*
qed

Every word represents some real

lemma *word-rep*:$\bigwedge bw$::*bword*. $\exists\, r$::*real*. *Rep-bword bw* $\equiv_E r$
proof −
fix *bw*
obtain *w* **where** *weq*:w = *Rep-bword bw* **by** *auto*
have *negInfCase*:w = *NEG-INF* $\Longrightarrow$ *?thesis bw*
apply(*rule exI*[**where** x=−((2 ^ 31) −1)])
using *weq* **by** (*auto simp add*: *repe.simps*)
have *posInfCase*:w = *POS-INF* $\Longrightarrow$ *?thesis bw*
apply(*rule exI*[**where** x=((2 ^ 31) −1)])
using *weq* **by** (*auto simp add*: *repe.simps*)
have *boundU*:$w \neq$ *NEG-INF* $\Longrightarrow w \neq$ *POS-INF* $\Longrightarrow$ *sint* (*Rep-bword bw*) < *sint POS-INF*
using *Rep-bword* [*of bw*]
by (*auto simp*: *less-le weq* [*symmetric*] *dest*: *sint-dist*)
have *boundL*:$w \neq$ *NEG-INF* $\Longrightarrow w \neq$ *POS-INF* $\Longrightarrow$ *sint NEG-INF* < *sint* (*Rep-bword bw*)
using *Rep-bword* [*of bw*]
by (*auto simp*: *less-le weq* [*symmetric*] *dest*: *sint-dist*)
have *intCase*:$w \neq$ *NEG-INF* $\Longrightarrow w \neq$ *POS-INF* $\Longrightarrow$ *?thesis bw*
apply(*rule exI*[**where** x= (*real-of-int* (*sint* (*Rep-bword bw*)))])
apply(*rule repINT*)
using *boundU boundL* **by**(*auto*)
then show *?thesis bw*
apply(*cases* w = *POS-INF*)
apply(*cases* w = *NEG-INF*)
using *posInfCase intCase negInfCase* **by** *auto*
qed

Every term has a value

lemma *eval-tot*:$(\exists\, r.\ ([\vartheta$::*trm*]$\nu' \downarrow r))$
proof (*induction* ϑ)
qed (*auto simp add*: *Min-def word-rep bword-neg-one-def*, *blast?*)

Interval formula semantics soundly implies real semantics

lemma *fml-sound*:
fixes φ::*formula* **and** ν::*wstate*
shows (*wfsem* φ ν *b*) $\Longrightarrow$ (ν *REP* ν') $\Longrightarrow$ (*rfsem* φ ν' *b*)
proof (*induction arbitrary*: ν' *rule*: *wfsem.induct*)
case (*wGreaterT t1 v t2 w*)
assume *wle*:*wgreater* (*fst* ([*t1*]<>*v*)) (*snd* ([*t2*]<>*v*))
assume *rep*:*v REP w*
obtain *r1* **and** *r2* **where** *eval1*:[*t1*]*w* ↓ *r1* **and** *eval2*:[*t2*]*w* ↓ *r2*
using *eval-tot*[*of t1 w*] *eval-tot*[*of t2 w*] **by** (*auto*)
note *rep1* = *trm-sound*[*of t1 w r1*, **where** ν=*v*, *OF eval1 rep*]
note *rep2* = *trm-sound*[*of t2 w r2*, **where** ν=*v*, *OF eval2 rep*]
show [*Greater t1 t2*]*w* ↓ *True*
apply(*rule rGreaterT*[**where** *?r1.0* = *r1*, **where** *?r2.0* = *r2*])
prefer *3*
apply(*rule wgreater-lemma*[**where** *?w1.0*=*fst*([*t1*]<> *v*), **where** *?w2.0*=*snd*([*t2*]<> *v*)])
using *rep1 rep2 wle repP-def repL-def repU-def eval1 eval2*
by ((*simp add*: *prod.case-eq-if* | *blast*)+)
next
case (*wGreaterF t2 v t1 v'*)
assume *wl*:*wgeq* (*fst* ([*t2*]<>*v*)) (*snd* ([*t1*]<>*v*))
assume *rep*:*v REP v'*
obtain *r1 r2*:: *real*
where *eval1*:(*rtsem t1 v' r1*) **and**
eval2:*rtsem t2 v' r2*
using *eval-tot*[*of t1 v'*] *eval-tot*[*of t2 v'*] **by** (*auto*)
note *rep1* = *trm-sound*[*of t1 v' r1*, **where** ν=*v*, *OF eval1 rep*]
note *rep2* = *trm-sound*[*of t2 v' r2*, **where** ν=*v*, *OF eval2 rep*]
show [*Greater t1 t2*]*v'* ↓ *False*
apply(*rule rGreaterF* [*of t1 v' r1 t2 r2*])
apply(*rule eval1*)
apply(*rule eval2*)
apply(*rule wgeq-lemma*[**where** *?w1.0*=*fst*([*t2*]<> *v*), **where** *?w2.0*=*snd*([*t1*]<> *v*)])
using *rep1 rep2 repP-def wgeq-lemma wl rep*
by *auto*
next
case (*wGeqT t1 v t2 v'*)
assume *a1*:*wgeq* (*fst* ([*t1*]<>*v*)) (*snd* ([*t2*]<>*v*))
assume *rep*:*v REP v'*
obtain *r1 r2*:: *real*
where *eval1*:(*rtsem t1 v' r1*) **and**
eval2:*rtsem t2 v' r2*
using *eval-tot*[*of t1 v'*] *eval-tot*[*of t2 v'*] **by** (*auto*)
note *rep1* = *trm-sound*[*of t1 v' r1*, **where** ν=*v*, *OF eval1 rep*]
note *rep2* = *trm-sound*[*of t2 v' r2*, **where** ν=*v*, *OF eval2 rep*]
show [*Geq t1 t2*]*v'* ↓ *True*
apply(*rule rGeqT*)

```
    apply(rule eval1)
   apply(rule eval2)
  using wgeq-lemma eval1 eval2 rep1 rep2 unfolding repP-def Let-def
  using wgreater-lemma prod.case-eq-if a1
  by auto
next
  case (wGeqF t2 v t1 v')
  assume a1:wgreater (fst ([t2]<>v)) (snd ([t1]<>v))
  assume rep:v REP v'
  obtain r1 r2:: real
  where eval1:(rtsem t1 v' r1) and
    eval2:rtsem t2 v'  r2
    using eval-tot[of t1 v'] eval-tot[of t2 v'] by (auto)
  note rep1 = trm-sound[of t1 v' r1, where ν=v, OF eval1 rep]
  note rep2 = trm-sound[of t2 v' r2, where ν=v, OF eval2 rep]
  show [Geq t1 t2]v' ⌊ False
   apply(rule rGeqF, rule eval1, rule eval2)
  using wgeq-lemma eval1 eval2 rep1 rep2 unfolding repP-def Let-def
  using wgreater-lemma rGreaterF prod.case-eq-if a1 rGreaterF by auto
next
  case (wEqualsT t2 v t1 v')
  assume eq1:fst ([t2]<>v) = snd ([t2]<>v)
  assume eq2:snd ([t2]<>v) = snd ([t1]<>v)
  assume eq3:snd ([t1]<>v) = fst ([t1]<>v)
  assume rep:v REP v'
  assume neq1:fst ([t2]<>v) ≠ NEG-INF
  assume neq2:fst ([t2]<>v) ≠ POS-INF
  obtain r1 r2:: real
  where eval1:(rtsem t1 v' r1) and
      eval2:rtsem t2 v'  r2
    using eval-tot[of t1 v'] eval-tot[of t2 v'] by (auto)
  note rep1 = trm-sound[of t1 v' r1, where ν=v, OF eval1 rep]
  note rep2 = trm-sound[of t2 v' r2, where ν=v, OF eval2 rep]
  show [Equals t1 t2]v' ⌊ True
   apply(rule rEqualsT, rule eval1, rule eval2)
   using eq1 eq2 eq3 eval1 eval2 rep1 rep2
   unfolding repP-def Let-def repL-def repU-def repe.simps using neq1 neq2 by
auto
next
  case (wEqualsF1 t1 v t2 v')
  assume wle:wgreater (fst ([t1]<>v)) (snd ([t2]<>v))
  assume rep:v REP v'
  obtain r1 r2:: real
  where eval1:(rtsem t1 v' r1) and
    eval2:rtsem t2 v'  r2
    using eval-tot[of t1 v'] eval-tot[of t2 v'] by (auto)
  note rep1 = trm-sound[of t1 v' r1, where ν=v, OF eval1 rep]
  note rep2 = trm-sound[of t2 v' r2, where ν=v, OF eval2 rep]
  show [Equals t1 t2]v' ⌊ False
```

apply(*rule rEqualsF*, *rule eval1*, *rule eval2*)
using *wgeq-lemma eval1 eval2 rep1 rep2 wgreater-lemma rGreaterF prod.case-eq-if wle*
unfolding *repP-def*
by (*metis* (*no-types*, *lifting*) *less-irrefl*)
next
case (*wEqualsF2 t2 v t1 v′*)
assume *wle*:*wgreater* (*fst* ([*t2*]<>*v*)) (*snd* ([*t1*]<>*v*))
assume *rep*:*v REP v′*
obtain *r1 r2*:: *real*
where *eval1*:(*rtsem t1 v′ r1*) **and**
eval2:*rtsem t2 v′ r2*
using *eval-tot*[*of t1 v′*] *eval-tot*[*of t2 v′*] **by** (*auto*)
note *rep1* = *trm-sound*[*of t1 v′ r1*, **where** *ν*=*v*, *OF eval1 rep*]
note *rep2* = *trm-sound*[*of t2 v′ r2*, **where** *ν*=*v*, *OF eval2 rep*]
show [*Equals t1 t2*]*v′* ⌊ *False*
apply(*rule rEqualsF*, *rule eval1*, *rule eval2*)
using *wgeq-lemma eval1 eval2 rep1 rep2 wgreater-lemma rGreaterF prod.case-eq-if wle*
unfolding *repP-def*
by (*metis* (*no-types*, *lifting*) *less-irrefl*)
qed (*auto*)

lemma *rep-upd*:*ω* = (*ν*(*Inr x* := *snd*([*ϑ*]<>*ν*)))(*Inl x* := *fst*([*ϑ*]<>*ν*)) ⟹ *ν REP ν′* ⟹ ([*ϑ*::*trm*]*ν′* ↓ *r*) ⟹ *ω REP ν′*(*x* := *r*)
apply(*rule REPI*)
apply(*rule conjI*)
apply(*unfold repL-def*)
using *trm-sound prod.case-eq-if repP-def repstate-simps repL-def*
apply(*metis* (*no-types*, *lifting*) *Inl-Inr-False fun-upd-apply sum.inject*(*1*))
using *repP-def repstate-simps repU-def*
apply(*auto simp add*: *repU-def*)
by (*metis* (*full-types*) *surjective-pairing trm-sound*)

Interval program semantics soundly contains real semantics existentially

theorem *interval-program-sound*:
fixes *α*::*prog*
shows ([[*α*]] *ν* ↓ *ω*) ⟹ *ν REP ν′* ⟹ (∃*ω′*. (*ω REP ω′*) ∧ ([*α*] *ν′* ⌊ *ω′*))
proof (*induction arbitrary*: *ν′ rule*: *wpsem.induct*)
case (*wTest φ ν ω ν′*)
assume *sem*:[[*φ*]]*ν* ↓ *True*
and *eq*:*ν* = *ω*
and *rep*:*ν REP ν′*
show *?case*
apply(*rule exI*[**where** *x*=*ν′*])
using *sem rep* **by** (*auto simp add*: *eq fml-sound rep*)
next
case (*wAssign ω ν x ϑ ν′*)
assume *eq*:*ω* = *ν*(*Inr x* := *snd* ([*ϑ*]<>*ν*), *Inl x* := *fst* ([*ϑ*]<>*ν*))

```
  and rep:ν REP ν'
  obtain r::real where eval:([ϑ::trm]ν' ↓ r) using eval-tot  by auto
  show ?case
    apply(rule exI[where x=ν'(x := r)])
    apply(rule conjI)
    apply(rule rep-upd[OF eq rep eval])
    apply auto
    apply(rule exI[where x=r])
    by (auto simp add: eval)
next case (wSeq α ν μ β ω ν') then show ?case by (simp, blast)
next case (wChoice1 a v w b v') then show ?case by auto
next case (wChoice2 a v w b v') then show ?case by auto
qed

end
```

References

[1] R. Bohrer. Differential dynamic logic. *Archive of Formal Proofs*, Feb. 2017. http://isa-afp.org/entries/Differential_Dynamic_Logic.html, Formal proof development.

[2] R. Bohrer, Y. K. Tan, S. Mitsch, M. O. Myreen, and A. Platzer. Veriphy: verified controller executables from verified cyber-physical system models. In J. S. Foster and D. Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 617–630. ACM, 2018.