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 **5**
1.1 Syntax . . . 5

2 Preliminary lemmas **8**
2.1 Case analysis lemmas . . . 8
2.2 Trivial arithmetic lemmas . . . 10

3 Arithmetic operations **11**
3.1 Addition upper bound . . . 11
3.2 Addition lower bound . . . 12
3.3 Max function . . . 12
3.4 Multiplication upper bound . . . 13
3.5 Exact multiplication . . . 14
3.6 Multiplication upper bound . . . 14
3.7 Minimum function . . . 14
3.8 Multiplication lower bound . . . 15
3.9 Negation . . . 17
3.10 Comparison . . . 17
3.11 Absolute value . . . 17

4 Finite Strings **18**

5 Syntax **23**

6 Semantics **24**

7 Soundness proofs **27**

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*⟩

lemma (**in** *linordered-idom*) *signed-less-neg-numeral-iff*:
‹*signed w* < − *numeral n* ⟷ *sint w* < − *numeral n*› (**is** ‹*?P* ⟷ *?Q*›)
⟨*proof*⟩

lemma (**in** *linordered-idom*) *numeral-less-signed-iff*:
‹*numeral n* < *signed w* ⟷ *numeral n* < *sint w*› (**is** ‹*?P* ⟷ *?Q*›)
⟨*proof*⟩

lemma (**in** *linordered-idom*) *neg-numeral-less-signed-iff*:
‹− *numeral n* < *signed w* ⟷ − *numeral n* < *sint w*› (**is** ‹*?P* ⟷ *?Q*›)
⟨*proof*⟩

lemma (**in** *linordered-idom*) *signed-nonnegative-iff*:
‹*0* ≤ *signed w* ⟷ *0* ≤ *sint w*› (**is** ‹*?P* ⟷ *?Q*›)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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 especially 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* = $\{n::word.\ sint\ n \geq sint\ NEG\text{-}INF \wedge sint\ n \leq sint\ POS\text{-}INF\}$
⟨*proof*⟩

Numeric literals

type-synonym *lit* = *bword*

setup-lifting *type-definition-bword*

lift-definition *bword-zero*::*bword* **is** *0*::*32 Word.word*
⟨*proof*⟩

lift-definition *bword-one*::*bword* **is** *1*::*32 Word.word*
⟨*proof*⟩

lift-definition *bword-neg-one*::*bword* **is** -1::*32 Word.word*
⟨*proof*⟩

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'*
⟨*proof*⟩

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′*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*}
⟨*proof*⟩

lemma *word-size-neg*:
fixes *w* :: *32 Word.word*
shows *size* (−*w*) = *size w*
⟨*proof*⟩

lemma *uint-distinct*:
fixes *w1 w2*
shows *w1* ≠ *w2* ⟹ *uint w1* ≠ *uint w2*
⟨*proof*⟩

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*:$l1 \leq u1$
assumes *ivl2*:$l2 \leq u2$
shows
$(l1 \leq 0 \wedge 0 \leq u1 \wedge l2 \leq 0 \wedge 0 \leq u2)$
$\vee(l1 \leq 0 \wedge 0 \leq u1 \wedge 0 \leq l2)$
$\vee(l1 \leq 0 \wedge 0 \leq u1 \wedge u2 \leq 0)$
$\vee(0 \leq l1 \wedge l2 \leq 0 \wedge 0 \leq u2)$
$\vee(u1 \leq 0 \wedge l2 \leq 0 \wedge 0 \leq u2)$
$\vee(u1 \leq 0 \wedge u2 \leq 0)$
$\vee(u1 \leq 0 \wedge 0 \leq l2)$
$\vee(0 \leq l1 \wedge u2 \leq 0)$
$\vee(0 \leq l1 \wedge 0 \leq l2)$
⟨*proof*⟩

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

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

$\Longrightarrow (w1 = NEG\text{-}INF \Longrightarrow w2 \neq POS\text{-}INF \Longrightarrow w2 \neq NEG\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow (w1 \neq POS\text{-}INF \Longrightarrow w1 \neq NEG\text{-}INF \Longrightarrow w2 = POS\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow (w1 \neq POS\text{-}INF \Longrightarrow w1 \neq NEG\text{-}INF \Longrightarrow w2 = NEG\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow (w1 \neq POS\text{-}INF \Longrightarrow w1 \neq NEG\text{-}INF \Longrightarrow w2 \neq POS\text{-}INF \Longrightarrow w2 \neq NEG\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow P\ w1\ w2$
⟨*proof*⟩

lemma *case-pu-inf*[*case-names PosAny AnyPos NegNeg NegNum NumNeg NumNum*]:
shows
$\bigwedge w1\ w2\ P.$
$(w1 = POS\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow (w2 = POS\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow (w1 = NEG\text{-}INF \Longrightarrow w2 = NEG\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow (w1 = NEG\text{-}INF \Longrightarrow w2 \neq POS\text{-}INF \Longrightarrow w2 \neq NEG\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow (w1 \neq POS\text{-}INF \Longrightarrow w1 \neq NEG\text{-}INF \Longrightarrow w2 = NEG\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow (w1 \neq POS\text{-}INF \Longrightarrow w1 \neq NEG\text{-}INF \Longrightarrow w2 \neq POS\text{-}INF \Longrightarrow w2 \neq NEG\text{-}INF \Longrightarrow P\ w1\ w2)$
$\Longrightarrow P\ w1\ w2$
⟨*proof*⟩

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

lemma *word-trichotomy*[*case-names Less Equal Greater*]:
fixes *w1 w2* :: *word*
shows
$(w1 <s w2 \Longrightarrow P\ w1\ w2) \Longrightarrow$
$(w1 = w2 \Longrightarrow P\ w1\ w2) \Longrightarrow$
$(w2 <s w1 \Longrightarrow P\ w1\ w2) \Longrightarrow P\ w1\ w2$
⟨*proof*⟩

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-INF ∧ w2 = POS-INF ⟹ P w1 w2*)
and *np*:(*w1 = NEG-INF ∧ w2 = POS-INF ⟹ P w1 w2*)
and *pn*:(*w1 = POS-INF ∧ w2 = NEG-INF ⟹ P w1 w2*)
and *nn*:(*w1 = NEG-INF ∧ w2 = NEG-INF ⟹ P w1 w2*)
and *pl*:(*w1 = POS-INF ∧ w2 ≠ NEG-INF ∧ w2 <s 0 ⟹ P w1 w2*)
and *ph*:(*w1 = POS-INF ∧ w2 ≠ POS-INF ∧ 0 <s w2 ⟹ P w1 w2*)
and *pz*:(*w1 = POS-INF ∧ w2 = 0 ⟹ P w1 w2*)
and *nl*:(*w1 = NEG-INF ∧ w2 ≠ NEG-INF ∧ w2 <s 0 ⟹ 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*⟩

2.2 Trivial arithmetic lemmas

lemma *max-diff-pos*:*0 ≤ 9223372034707292161 + ((−(2 ^ 31))::real)* ⟨*proof*⟩

lemma *max-less*:*2 ^ 31 < (9223372039002259455::int)* ⟨*proof*⟩

lemma *sints64*:*sints 64 = {i. − (2 ^ 63) ≤ i ∧ i < 2 ^ 63}*
⟨*proof*⟩

lemma *sints32*:*sints 32 = {i. − (2 ^ 31) ≤ i ∧ i < 2 ^ 31}*
⟨*proof*⟩

lemma *upcast-max*:*sint((scast(0x80000001::word))::64 Word.word)=sint((0x80000001::32 Word.word))*
⟨*proof*⟩

lemma *upcast-min*:*(0xFFFFFFFF80000001::64 Word.word) = ((scast (−0x7FFFFFFF::word))::64 Word.word)*
⟨*proof*⟩

lemma *min-extend-neg*:*sint ((0xFFFFFFFF80000001)::64 Word.word) < 0*
⟨*proof*⟩

lemma *min-extend-val'*:*sint ((−0x7FFFFFFF)::64 Word.word) = (−0x7FFFFFFF)*
⟨*proof*⟩

lemma *min-extend-val*:$(-0x7FFFFFFF::64\ Word.word) = 0xFFFFFFFF80000001$
⟨*proof*⟩

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

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*)
⟨*proof*⟩

lemma *pu-lemma*:
fixes *w1 w2*

fixes *r1 r2 :: real*
assumes *up1:w1* $\equiv_U$ *(r1::real)*
assumes *up2:w2* $\equiv_U$ *(r2::real)*
shows *pu w1 w2* $\equiv_U$ *(r1 + r2)*
⟨*proof*⟩

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* $\equiv_L$ *(r1::real)*
assumes *lo2:w2* $\equiv_L$ *(r2::real)*
shows *pl w1 w2* $\equiv_L$ *(r1 + r2)*
⟨*proof*⟩

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* $\equiv_E$ *(r1::real)*

assumes *eq2*:*w2* $\equiv_E$ (*r2*::*real*)
shows *wmax w1 w2* $\equiv_E$ (*max r1 r2*)
⟨*proof*⟩

lemma *max-repU1*:
assumes *w1* $\equiv_U$ *x*
assumes *w2* $\equiv_U$ *y*
shows *wmax w1 w2* $\equiv_U$ *x*
⟨*proof*⟩

lemma *max-repU2*:
assumes *w1* $\equiv_U$ *y*
assumes *w2* $\equiv_U$ *x*
shows *wmax w1 w2* $\equiv_U$ *x*
⟨*proof*⟩

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*⟩

Product of 32-bit numbers fits in 64 bits

lemma *times-upcast-upper*:
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 $x * y \leq 4611686018427387904$
⟨*proof*⟩

Correctness of 32x32 bit multiplication

3.5 Exact multiplication

lemma *wtimes-exact*:
assumes *eq1*:$w1 \equiv_E r1$
assumes *eq2*:$w2 \equiv_E r2$
shows $wtimes\ w1\ w2 \equiv_E r1 * r2$
⟨*proof*⟩

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*:$u_1 \equiv_U (r1::real)$
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 $tu\ l_1\ u_1\ l_2\ u_2 \equiv_U (r1 * r2)$
⟨*proof*⟩

3.7 Minimum function

Minimum of 2s-complement words

fun *wmin* :: *word* ⇒ *word* ⇒ *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*⟩

lemma *min-repU1*:
assumes *w1* $\equiv_L$ *x*
assumes *w2* $\equiv_L$ *y*
shows *wmin w1 w2* $\equiv_L$ *x*
⟨*proof*⟩

lemma *min-repU2*:
assumes *w1* $\equiv_L$ *y*
assumes *w2* $\equiv_L$ *x*
shows *wmin w1 w2* $\equiv_L$ *x*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*))
⟨*proof*⟩

2s-complement encoding is injective

lemma *sint-dist*:
fixes *x y* ::*word*
assumes *x* ≠ *y*
shows *sint x* ≠ *sint y*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

Comparison < is correct

lemma *wgreater-lemma*:*w1* ≡$_L$ (*r1*::*real*) ⟹ *w2* ≡$_U$ *r2* ⟹ *wgreater w1 w2* ⟹ *r1* > *r2*
⟨*proof*⟩

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* ≡$_L$ *r1* ⟹ *w2* ≡$_U$ (*r2*::*real*) ⟹ *wgeq w1 w2* ⟹ *r1* ≥ *r2*
⟨*proof*⟩

3.11 Absolute value

Absolute value of word

fun *wabs* :: *word* ⇒ *word*
where *wabs l1* = (*wmax l1* (*wneg l1*))

Correctness of wmax

lemma *wabs-lemma*:
assumes *eq*:*w* ≡$_E$ (*r*::*real*)
shows *wabs w* ≡$_E$ (*abs r*)
⟨*proof*⟩

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*
  ⟨*proof*⟩

Lift definition of string length

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

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*
  ⟨*proof*⟩

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

Finite strings are finite

instantiation *fin-string* :: *finite* **begin**
instance ⟨*proof*⟩
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* ⟨*proof*⟩
lift-definition *less-fin-string*::*fin-string* ⇒ *fin-string* ⇒ *bool* **is** *less-charlist* ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *lleq-less*:
  fixes *x y*
  shows (*less-charlist x y*) = (*lleq-charlist x y* ∧ ¬ *lleq-charlist y x*)
  ⟨*proof*⟩

lemma *lleq-refl*:
  fixes *x*
  shows *lleq-charlist x x*
  ⟨*proof*⟩

lemma *lleq-trans*:

fixes *x y z*
shows *lleq-charlist x y* $\Longrightarrow$ *lleq-charlist y z* $\Longrightarrow$ *lleq-charlist x z*
⟨*proof*⟩

lemma *lleq-antisym*:
fixes *x y*
shows *lleq-charlist x y* $\Longrightarrow$ *lleq-charlist y x* $\Longrightarrow$ $x = y$
⟨*proof*⟩

lemma *lleq-dichotomy*:
fixes *x y*
shows *lleq-charlist x y* $\vee$ *lleq-charlist y x*
⟨*proof*⟩

instance
⟨*proof*⟩
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* $\Rightarrow$ *string* $\Rightarrow$ *string*
where *string-cons c s* = (*if length s* $\geq$ *MAX-STR then s else c* # *s*)

lift-definition *fin-string-empty*::*fin-string* **is** ″″ ⟨*proof*⟩
lift-definition *fin-string-cons*::*char* $\Rightarrow$ *fin-string* $\Rightarrow$ *fin-string* **is** *string-cons* ⟨*proof*⟩
lift-definition *fin-string-expose*::*fin-string* $\Rightarrow$ (*unit* + (*char*$*$*fin-string*)) **is** *string-expose*

⟨*proof*⟩

Helper functions for enum typeclass instance

fun *fin-string-upto* :: *nat* $\Rightarrow$ *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* $\Longrightarrow$ *List.member* (*L1* @ *L2*) *x*
⟨*proof*⟩

lemma *mem-appR*:*List.member L2 x* $\Longrightarrow$ *List.member* (*L1* @ *L2*) *x*
⟨*proof*⟩

lemma *mem-app-or*:*List.member* (*L1* @ *L2*) *x* = *List.member L1 x* $\vee$ *List.member L2 x*
⟨*proof*⟩

lemma *fin-string-nil*:
fixes *n*
shows *List.member* (*fin-string-upto n*) *fin-string-empty*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *enum-chars*:*set* (*enum-class.enum*::*char list*)= *UNIV*
⟨*proof*⟩

lemma *member-concat*:*List.member* (*concat LL*) *x* = (∃ *L*. *List.member LL L* ∧ *List.member L x*)
⟨*proof*⟩

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*⟩

lemma *fin-string-upto-length*:
shows *List.member* (*fin-string-upto n*) *L* ⟹ *ilength L* ≤ *n*
⟨*proof*⟩

fin-string-upto produces no duplicate identifiers

lemma *distinct-upto*:
shows *i* ≤ *MAX-STR* ⟹ *distinct* (*fin-string-upto i*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *vals-ALL*:*set* (*vals*::*fin-string list*) = *UNIV*
⟨*proof*⟩

lemma *setA*:
assumes *set*:⋀*y*. *y* ∈ *set L* ⟹ *P y*
shows *list-all P L*
⟨*proof*⟩

lemma *setE*:
assumes *set*: *y* ∈ *set L*
assumes *P*:*P y*
shows *list-ex P L*
⟨*proof*⟩

instance
⟨*proof*⟩
end

instantiation *fin-string* :: *equal* **begin**
definition *equal-fin-string* :: *fin-string* ⇒ *fin-string* ⇒ *bool*
where [*code*]:*equal-fin-string X Y* = (*X* ≤ *Y* ∧ *Y* ≤ *X*)
instance
⟨*proof*⟩
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* ϑ ϑ′ = (*Not* (*Geq* ϑ ϑ′))

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* ν ≡ (⋀*i*. *word* (ν (*Inl i*)) ∧ *word* (ν (*Inr i*)))

Interpretation of a term in a state

inductive *rtsem* :: *trm* ⇒ *rstate* ⇒ *real* ⇒ *bool* (‹([-]- ↓ -)› *10*)
where
rtsem-Const:*Rep-bword w* ≡$_E$ *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** ‹≥$_S$› *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* ϑ1 ϑ2]<> ν) =
(*let* (*l1*, *u1*) = [ϑ1]<> ν *in*
let (*l2*, *u2*) = [ϑ2]<> ν *in*
(*pl l1 l2*, *pu u1 u2*))
| *wTimesU*:([(*Times* ϑ1 ϑ2)]<> ν) =
(*let* (*l1*, *u1*) = [ϑ1]<> ν *in*
let (*l2*, *u2*) = [ϑ2]<> ν *in*
(*tl l1 u1 l2 u2*, *tu l1 u1 l2 u2*))
| *wMaxU*:([(*Max* ϑ1 ϑ2)]<> ν) =
(*let* (*l1*, *u1*) = [ϑ1]<> ν *in*
let (*l2*, *u2*) = [ϑ2]<> ν *in*
(*wmax l1 l2*, *wmax u1 u2*))
| *wMinU*:([(*Min* ϑ1 ϑ2)]<> ν) =
(*let* (*l1*, *u1*) = [ϑ1]<> ν *in*
let (*l2*, *u2*) = [ϑ2]<> ν *in*
(*wmin l1 l2*, *wmin u1 u2*))
| *wNegU*:([(*Neg* ϑ)]<> ν) =
(*let* (*l*, *u*) = [ϑ]<> ν *in*
(*wneg u*, *wneg l*))
| *wAbsU*:([(*Abs* ϑ1)]<> ν) =
(*let* (*l1*, *u1*) = [ϑ1]<> ν *in*
(*wmax l1* (*wneg u1*), *wmax u1* (*wneg l1*)))

inductive *wfsem* :: *formula* ⇒ *wstate* ⇒ *bool* ⇒ *bool* (‹([[-]]- ↓ -)› *20*)
where
wGreaterT:*wgreater* (*fst* ([ϑ1]<>ν)) (*snd* ([ϑ2]<>ν)) ⟹ ([[(*Greater* ϑ1 ϑ2)]]ν ↓ *True*)
| *wGreaterF*:*wgeq* (*fst* ([ϑ2]<>ν)) (*snd* ([ϑ1]<>ν)) ⟹ ([[(*Greater* ϑ1 ϑ2)]]ν ↓ *False*)
| *wGeqT*:*wgeq* (*fst* ([ϑ1]<> ν)) (*snd* ([ϑ2]<>ν)) ⟹ ([[(*Geq* ϑ1 ϑ2)]]ν ↓ *True*)
| *wGeqF*:*wgreater* (*fst* ([ϑ2]<>ν)) (*snd* ([ϑ1]<>ν)) ⟹ ([[(*Geq* ϑ1 ϑ2)]]ν ↓ *False*)
| *wEqualsT*:⟦(*fst* ([ϑ2]<>ν) = *snd* ([ϑ2]<>ν)); (*snd* ([ϑ2]<>ν) = *snd* ([ϑ1]<>ν));
(*snd* ([ϑ1]<>ν) = *fst* ([ϑ1]<>ν)); (*fst* ([ϑ2]<>ν) ≠ *NEG-INF*);
(*fst* ([ϑ2]<>ν) ≠ *POS-INF*)⟧
⟹ ([[*Equals* ϑ1 ϑ2]] ν ↓ *True*)
| *wEqualsF1*:*wgreater* (*fst* ([ϑ1]<> ν)) (*snd* ([ϑ2]<>ν)) ⟹ ([[*Equals* ϑ1 ϑ2]] ν ↓ *False*)
| *wEqualsF2*:*wgreater* (*fst* ([ϑ2]<> ν)) (*snd* ([ϑ1]<>ν)) ⟹ ([[*Equals* ϑ1 ϑ2]] ν ↓ *False*)
| *wAndT*:⟦[[φ]]ν ↓ *True*; [[ψ]]ν ↓ *True*⟧ ⟹ ([[*And* φ ψ]]ν ↓ *True*)
| *wAndF1*:[[φ]]ν ↓ *False* ⟹ ([[*And* φ ψ]]ν ↓ *False*)
| *wAndF2*:[[ψ]]ν ↓ *False* ⟹ ([[*And* φ ψ]]ν ↓ *False*)
| *wOrT1*:([[φ]]ν ↓ *True*) ⟹ ([[*Or* φ ψ]]ν ↓ *True*)
| *wOrT2*:([[ψ]]ν ↓ *True*) ⟹ ([[*Or* φ ψ]]ν ↓ *True*)

| *wOrF*:⟦[[[φ]]ν ↓ *False*; [[ψ]]ν ↓ *False*⟧ ⟹ ([[*And* φ ψ]]ν ↓ *False*)
| *wNotT*:([[φ]]ν ↓ *False*) ⟹ ([[*Not* φ]]ν ↓ *True*)
| *wNotF*:([[φ]]ν ↓ *True*) ⟹ ([[*Not* φ]]ν ↓ *False*)

inductive-simps
wfsem-Gr-simps[*simp*]: ([[*Le* ϑ_1 ϑ_2]]ν ↓ *b*)
and *wfsem-And-simps*[*simp*]: ([[*And* φ ψ]]ν ↓ *b*)
and *wfsem-Or-simps*[*simp*]: ([[*Or* φ ψ]]ν ↓ *b*)
and *wfsem-Not-simps*[*simp*]: ([[*Not* φ]]ν ↓ *b*)
and *wfsem-Equals-simps*[*simp*]: ([[*Equals* ϑ_1 ϑ_2]]ν ↓ *b*)

Program semantics

inductive *wpsem* :: *prog* ⇒ *wstate* ⇒ *wstate* ⇒ *bool* (‹([[-]]- ↓ -)› *20*)
where
wTest:([[φ]]ν ↓ *True*) ⟹ ν = ω ⟹ ([[? φ]] ν ↓ ω)
| *wSeq*:([[α]]ν ↓ μ) ⟹ ([[β]] μ ↓ ω) ⟹ ([[α;; β]] ν ↓ ω)
| *wAssign*:ω = ((ν ((*Inr x*) := *snd*([ϑ]<>ν))) ((*Inl x*) := *fst*([ϑ]<>ν))) ⟹ ([[*Assign* *x* ϑ]] ν ↓ ω)
| *wChoice1*[*simp*]:([[α]]ν ↓ ω) ⟹ ([[*Choice* α β]]ν ↓ ω)
| *wChoice2*[*simp*]:([[β]]ν ↓ ω) ⟹ ([[*Choice* α β]]ν ↓ ω)

inductive-simps
wpsem-Test-simps[*simp*]: ([[*Test* φ]]ν ↓ *b*)
and *wpsem-Seq-simps*[*simp*]: ([[α;; β]]ν ↓ *b*)
and *wpsem-Assign-simps*[*simp*]: ([[*Assign* *x* ϑ]]ν ↓ *b*)
and *wpsem-Choice-simps*[*simp*]: ([[*Choice* α β]]ν ↓ *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* ⇒ *rstate* ⇒ *bool* (**infix** ‹*REP*› *10*)
where *REPI*:(⋀*x*. (ν (*Inl x*) ≡$_L$ ν' *x*) ∧ (ν (*Inr x*) ≡$_U$ ν' *x*)) ⟹ (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]ν' ↓ *r*) ⟹ (ν *REP* ν') ⟹ ([ϑ]<>ν) ≡$_P$ *r*
⟨*proof*⟩

Every word represents some real

lemma *word-rep*:⋀*bw*::*bword*. ∃ *r*::*real*. *Rep-bword bw* ≡$_E$ *r*
⟨*proof*⟩

Every term has a value

lemma *eval-tot*:$(\exists\, r.\ ([\vartheta{::}trm]\nu' \downarrow r))$
⟨*proof*⟩

Interval formula semantics soundly implies real semantics

lemma *fml-sound*:
  fixes $\varphi{::}formula$ **and** $\nu{::}wstate$
  shows $(wfsem\ \varphi\ \nu\ b) \Longrightarrow\ (\nu\ REP\ \nu') \Longrightarrow (rfsem\ \varphi\ \nu'\ \ b)$
⟨*proof*⟩

lemma *rep-upd*:$\omega = (\nu(Inr\ x := snd([\vartheta]{<}{>}\nu)))(Inl\ x := fst([\vartheta]{<}{>}\nu))$
$\Longrightarrow \nu\ REP\ \nu' \Longrightarrow ([\vartheta{::}trm]\nu' \downarrow r) \Longrightarrow \omega\ REP\ \nu'(x := r)$
  ⟨*proof*⟩

Interval program semantics soundly contains real semantics existentially

theorem *interval-program-sound*:
  fixes $\alpha{::}prog$
  shows $([[\alpha]]\ \nu \downarrow \omega)\ \Longrightarrow \nu\ REP\ \nu' \Longrightarrow (\exists\omega'.\ (\omega\ REP\ \omega') \wedge ([\alpha]\ \nu' \downarrow \omega'))$
⟨*proof*⟩

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.