Formally Verified Interval Arithmetic and Its Application to Program Verification

Achim D. Brucker
Teddy Cameron-Burke
Amy Stell
a.brucker@exeter.ac.uk
tc599@exeter.ac.uk
a.stell@exeter.ac.uk
The University of Exeter
Department of Computer Science
Exeter, Devon, UK

ABSTRACT

Interval arithmetic is a well known mathematical technique to analyse or mitigate rounding or measurement errors. Thus, it is promising to integrate interval analysis into program verification environments. Such an integration is not only useful for the verification of numerical algorithms: the need to ensure that computations stay within certain bounds is common. For example, to show that arithmetic computations stay within the hardware bounds of a given number representation.

We present a formalisation of (extended) interval arithmetic in Isabelle/HOL, including the concept of inclusion isotone (extended) interval analysis. The main result of this part is the formal proof that interval-splitting converges for Lipschitz-continuous interval isotone functions. We also show how interval types can be used for verifying programs in a C-like programming language.

CCS CONCEPTS

• Theory of computation → Semantics and reasoning; Interactive proof systems;
• Software and its engineering → Software notations and tools;
• Mathematics of computing → Mathematical analysis; Computing methodologies → Symbolic and algebraic manipulation.

KEYWORDS

Extended Interval Analysis, Program Verification, Formalising Mathematics, Isabelle/HOL

1 INTRODUCTION

Interval analysis [28] in general and, in particular, inclusion isotone (or inclusion isotonic) extended interval arithmetic [34], are well known mathematical techniques to analyse or mitigate rounding errors or measurement errors [13]. Thus, it is promising to integrate interval analysis into program verification environments. Such an integration is not only useful for the verification of numerical algorithms: the need to ensure that computations stay within certain bounds is common. For example, to show that computations stay within the hardware bounds of a given number representation.

Another application is the verification of cyber-physical systems, where a discretised implementation approximates a system described in physical quantities expressed using perfect mathematical reals, and perfect ordinary differential equations. Moreover, first applications of interval analysis to the security analysis of neural networks are promising [38, 17].

In this paper, we present a formalisation of interval arithmetic in Isabelle/HOL, including the concept of inclusion isotone (extended) interval analysis. Moreover, we show how interval types can be integrated into the weakest precondition generation of Clean programs and demonstrate the use of interval arithmetic in program verification. In more detail, our contributions are:

(1) a conservative formalisation of (extended) interval arithmetic in Isabelle/HOL, including interval extension of functions (Sect. 3.1), inclusion isotonicity (Sect. 3.2), Lipschitz continuity (Sect. 3.3), division of intervals (Sect. 3.4), extended interval arithmetic (Sect. 3.5), and multi-intervals (Sect. 3.6). The main result of this part is the formal proof that interval-splitting converges for Lipschitz-continuous inclusion isotone functions at least linearly with the number of splits (Sect. 3.7);

(2) an integration of the extended interval type into Clean, allowing for using interval analysis as part of the verification of imperative programs (Sect. 4);

(3) a case study, demonstrating the use of interval arithmetic in program verification (Sect. 5).

We continue with introducing technical background of our work in Sect. 2. Thereafter, we present the main contributions of our paper (Sect. 3-5). Finally, we discuss related work (Sect. 6) and draw conclusions (Sect. 7).
2 BACKGROUND

In this section, we introduce the technical background of our work, namely Isabelle/HOL [32], Isabelle’s interval type, and Clean [36].

2.1 Isabelle/HOL

Isabelle [32] is an interactive theorem prover that has been used successfully in large-scale verification projects (e.g., [23] presents the verification of an operating system kernel using Isabelle/HOL). The formal language of Isabelle is Higher-order logic (HOL) [10], which is a classical logic based on a simple type system. It provides the usual logical connectives like \( \neg \), \( \land \), \( \lor \), \( \lor \rightarrow \) as well as the object-logical quantifiers \( \forall \ x. \ P \ x \) and \( \exists \ x. \ P \ x \). In contrast to first-order logic, quantifiers may range over arbitrary types, including total functions \( f::\alpha \Rightarrow \beta \) (where \( \alpha \) and \( \beta \) are polymorphic type variables).

Isabelle/HOL allows for extending theories in a logically safe way: A theory-extension is conservative if the extended theory is consistent, provided that the original theory was consistent. Conservative extensions can be constant definitions, type definitions, datatype definitions, primitive recursive definitions and well-founded recursive definitions.

2.2 Interval Arithmetic

Isabelle [32] provides a formalisation of (closed) intervals in the theory HOL-Library.Interval. This theory provides a datatype modelling intervals as tuples, where the first element is the lower bound of the interval and the second element is the upper bound:

```isar
typedef (overloaded) 'a interval = (\':a::preorder, b) a \leq b
morphisms bounds_of_interval Interval by auto
```

The interval theory defines the basic operations on intervals such as subtraction, addition, and multiplication (but not division). It also shows that intervals are an instantiation of certain type classes (e.g., preorder, equal and lattice). However, it does not define interval arithmetic in the way mathematicians typically understand it, nor does it include the property of inclusion isotonicity, or the notion of interval extension and Lipschitz continuity on intervals.

One notable aspect of the existing theory is that it leaves an interval undefined if the upper bound is lower than the lower bound. To address this, we developed a definition `mk_interval`, which swaps the bounds if this condition is met, ensuring the proper definition of an interval:

```isar
definition mk_interval where
  mk_interval = (\'a (x,y). Interval (if x \leq y then (x,y) else (y,x)))
```

In examples, we might use the mathematical notation \([l,u]\) for `Interval (1, u)`. Furthermore, we also prove the following equality, which often is suitable as a simplification rule in proofs:

```isar
lemma mk_interval':
  mk_interval = (\'a (x,y). (if x \leq y then Interval(x, y) else Interval(y, x)))
unfolding mk_interval_def by(rule ext, rule arg_cong, auto)
```

2.2.1 Convergence Example. Interval arithmetic is often used in computer programs to mitigate rounding errors [13], or ensure that computations stay within hardware bounds. However, intervals often produce an over approximation which creates a degree of uncertainty. This over approximation, can potentially compromise the assurance of safety requirements being met.

To mitigate this, a technique known as splitting can be utilised. This method involves subdividing the interval into smaller segments, thereby reducing the extent of over approximation. As the process of splitting is performed, the approximation converges towards the precise value. This increases the precision of the computations and can ensure the adherence to safety requirements. For example, let us consider the function

\[
f(x) = 2x - x
\]

We know this is an inclusion isitone, Lipschitz continuous function. For the values \( x \in \{0..1\} \), we can easily verify that the bounds when the function \( f \) is applied are \( \{0..1\} \) (e.g. \( f(0.1) = \{0..1\} \)).

The interval extension of this function is:

\[
F(X) = ([2,2] \cdot X) - X
\]

where \([2,2]\) denotes the (closed) interval. When we apply the extension to the interval \([0,1]\) we get \( F([0,1]) = [-1,2] \), hence its excess width is \( 3 - 1 = 2 \).

Now let us split \([0,1]\) into \( N \) subintervals, we then take the union of the resulting subintervals and find narrower interval bounds that still contain the range of the values of \( f(X) \). Table 1 shows the union of the results for a range of \( N \).

| \( N \) | Result \( f(X) \) | Width \( |f(X)| \) |
|---|---|---|
| 1 | \([-1,000,2,000]\) | 3,000 |
| 2 | \([-0,500,1,500]\) | 2,000 |
| 10 | \([-0,100,1,100]\) | 1,200 |
| 100 | \([-0,010,1,010]\) | 1,020 |
| 1,000 | \([-0,001,1,001]\) | 1,002 |

Table 1: Demonstrating the convergence (evidenced by the width of the resulting interval) of the refinement of the function \( F(X) = ([2,2] \cdot X) - X \) when splitting the interval \([0,1]\) into \( N \) subdivisions.

2.3 Clean

Clean [36] is a small imperative C-style language in the spirit of Winskel’s IMP [39] and its formalisation in Isabelle/HOL [31]. In contrast to most other embeddings of programming languages into Isabelle/HOL that are deep embeddings (e.g., [31, 24, 25, 35]), Clean is a shallow embedding (for a discussion of deep and shallow embeddings, we refer the reader to [5, 1]). Clean supports a type-safe notion of program-variables, an incremental construction of the typed state-space, support of incremental verification, and open-world extensibility of new type definitions. The Clean language supports conditionals and loops as well as break and return.

To define a (non-recursive) Clean function, we use the Isar command `function_spec` detailing the function name, return type, precondition and postconditions, local variables, and body. For example, the well known `weighted sum` algorithm:
Definition 2.1 (Weighted Sum). A weighted sum over some SemiRing \((T, +, \cdot)\) is a linear combination of vectors \(\mathbf{x} = (x_1, \ldots, x_n)\) and weights \(\mathbf{w} = (w_1, \ldots, w_n)\)

\[
\text{weighted_sum}(\mathbf{x}, \mathbf{w}) = \sum_{i=1}^{n} x_i \cdot w_i
\]

would be expressed in Clean as:

```clean
function_spec weighted_sum(xs :: ('a list),
ws :: ('a list)) returns ('a)
pre (length xs = length ws)
post res. res = (sum_list (map2 (*) xs ws));
local_vars
i : (nat)
out : ('a)
defines!
out := 0 ;;
(i := 0) ;;
while (i < length xs) do
   (out := out + xs ! i * ws ! i) ;;
   (i := i + 1) ;;
ed ;;
return result_value_update (out)
```

Hence, when needed we can replace the polymorphic type `'a` with any type which instantiates addition and multiplication. The consequence of this abstraction is the ability to formalise generic programs and emulate a more extensible program design without the need to expand the base semantics.

When considering the total correctness of the weighted sum function, Clean will generate precondition and postcondition definitions from their respective keywords. In Isabelle, we express this correctness using the Hoare Triple:

```isabelle
lemma weighted_sum_correct:
  \{ \sigma. \sigma \land \text{weighted_sum_pre (xs, ws)} \} \sigma \land \sigma = \sigma_w
  \{ \text{weighted_sum (xs, ws)} \}
  \{ \text{res. \sigma \land \text{weighted_sum_post (xs, ws)} \} \sigma_w = \sigma \land \sigma_res \}
```

The \(\sigma \Rightarrow \sigma\) asserts that the state \(\sigma\) is free of break and return state changes. Also note that the verification is solely focusing on total correctness, thus to verify this function would require a measure alongside the invariant.

3 FORMALISING INTERVAL ARITHMETIC

In this section we present our formalisation of interval arithmetic and extended interval analysis. For the terms and definitions discussed we mostly follow the notation presented by Moore et al in [28], but we will note where our definitions deviate.

3.1 Interval Extensions of Functions

In the context of interval arithmetic, a function \(F\) is considered an interval extension of a function \(f\) under the condition that for any degenerate interval argument (e.g., an interval that consists of a single point), \(F\) yields the same result as \(f\). This can be expressed as \(F([x, x]) = f(x)\). In Isabelle, we define this as the following:

```isabelle
definition extend_natural where
extend_natural f = (\lambda X. mk_interval (f (lower X), f (upper X)))
```

We choose to keep the definitions as generic as possible by using type variables (e.g., `'a`, `'b`), instead of concrete types (e.g., `real`, `int`), for functions and their results. By choosing appropriate `'a` and `'b` we can select a representation which is appropriate for the problem domain.

Moore et al [28] define a more general natural interval extension to follow from these properties:

1. the real variable \(x\) is replaced by an interval \(X\);
2. the real arithmetic operations are replaced with the corresponding interval ones.

If these properties are met then \(F\) is a natural extension of \(f\).

The formalisation of intervals in Isabelle uses a shallow embedding and so does our work on (extended) interval arithmetic. By using a shallow embedding, this prevents us from replacing real arithmetic operations to interval arithmetic operations in a straightforward way. More importantly, the definition of Moore et al [28] assumes that the operations of intervals enjoy essentially the same properties (like commutativity, associativity) than their counterparts over reals (and essentially have a very similar semantics). This is something that we need to prove formally in Isabelle. Furthermore, our formalisation is more generic in the sense that it does not only work over real intervals.

Hence, we define our stronger version of interval extension by first applying the function \(f\) onto both the upper and lower bounds of the interval, and then constructing a new interval with the bounds of the result. This is particularly useful when evaluating bounds on real valued functions:

```isabelle
definition is_interval_extension_of where
(is_interval_extension_of f) =
(\forall x. (f (interval_of x)) = interval_of (f x))
```

In interval analysis it is important to be able to compute the interval extension function \(F\) from the function \(f\). To this extent, the following definition, extend natural, takes a function \(f\) and returns a new function that, given an interval \(X\), returns a new interval constructed from the lower and upper bounds of \(X\) evaluated through \(f\). This allows us to extend the domain of \(f\) to intervals.

```isabelle
definition extend natural where
extend natural f = (\lambda X. mk_interval (f (lower X), f (upper X)))
```

To verify the correctness of extend natural, we have proven two lemmas. The first lemma shows that extend natural \(f\) is an interval extension of \(f\), and the second lemma shows that extend natural \(f\) is a natural interval extension of \(f\). These lemmas are stated and proven as follows:

```isabelle
definition is_interval_extension_of where
(is_interval_extension_of f) =
(\forall x. (f (interval_of x)) = interval_of (f x))
```
These provide a formal guarantee that `extend_natural` correctly extends a function to intervals according to our definitions of interval and natural interval extensions.

### 3.2 Inclusion Isotonicity

Inclusion isotonicity ensures that the relationship between intervals is preserved under the extension of functions.

A function is said to be inclusion isotope if for any \( i = 0, \ldots, n - 1 \) and \( Y_i \leq X_i \) implies

\[
F(Y_0, \ldots, Y_{n-1}) \leq F(X_0, \ldots, X_{n-1})
\]  

(4)

This means that if the upper and lower bounds of the interval \( Y_i \) are contained within the bounds of the interval \( X_i \) then the output of the function \( F \) applied to \( Y_i \) is contained within the output of the function \( F \) applied to \( X_i \).

As \([0, 2] \leq [−4, 5]\) we know the result of the \( Y_1 = [2, 2] \cdot [0, 2] − [0, 2] = [−4, 5] \) is contained within the result of the \( Y_2 = [2, 2] \cdot [−4, 5] = [−13, 14] \), and hence \( Y_1 \leq Y_2 \) as \([−4, 5] \leq [−13, 14] \).

We define the property of inclusion isotonicity as follows:

Note that in terms of intervals the less than equal inequality, \( \leq \), is the equivalent of a subset equal relationship, \( \subseteq \), on sets.

It then easily follows that for all affine functions of type real to real, if we use our function `extend_natural` (discussed in Sect. 3.1) to extend this to a function that can work on the interval domain, then the extension is also inclusion isotope.

This leads us to the fundamental theory of interval analysis, (theorem 5.1 in [28]) which states that if \( F \) is an inclusion isotonic, interval extension of \( f \), then

\[
f(x_0, \ldots, x_{n-1}) \subseteq \text{set_of}(F(x_0, \ldots, x_{n-1})).
\]  

(5)

In Isabelle/HOL this lemma is stated as the following:

Where \( F \) (set_of \( X \)) is the image of the set representing the interval \( X \) (i.e., set_of \( X = (\text{lower } X \ldots \text{upper } X) \)) under the application of \( f \).

The proof for this lemma follows from the definition of interval extension, which states that \( \forall x. F([x, x]) = f(x) \) (see Sect. 3.1), and the definition of inclusion isotope (see Equation 4).

There are slight differences in representation of our lemma fundamental_theorem_of_interval_analysis, as, while Moore et al [28] use a representation of intervals synonymous with set notation, the strict type system of Isabelle/HOL requires the explicit conversion between the interval and set datatypes (using set_of).

### 3.3 Lipschitz Interval Extension

The combination of inclusion isotonicity, with Lipschitz continuity provides a reliable enclosure of the range of a function over any interval. This is because Lipschitz continuity ensures that there are no abrupt changes in a function that could lead to holes in the interval extension.

More formally, in the context of real-valued functions, a function \( f : \mathbb{R} \rightarrow \mathbb{R} \) is Lipschitz continuous if there exists a positive real constant \( C \) such that, for all real \( x \) and \( y \), the absolute difference in the function values at these points is less than or equal to \( C \) times the absolute difference in the points. In the theory HOL-Analysis.Lipschitz [32] this is defined as:

We extend the notion of Lipschitz continuity to interval arithmetic, so that a function \( F \) is said to be Lipschitz continuous on an interval if there exists a real constant \( 0 \leq C \) such that, the width of the result of the function \( F \) applied onto the interval \( X \), is less than or equal to the width of the original interval \( X \) multiplied by some constant \( C \). This constant \( C \) is referred to as the Lipschitz constant for the function. We define this Lipschitz interval extension as the following in Isabelle/HOL:

From the definition of Lipschitz continuity on intervals we know that the width of \( F(X) \) approaches zero at least linearly with the width of \( X \). This is an important idea which will be discussed in more detail in Sect. 3.7 to closely approximate \( f(x_0, \ldots, x_{n-1}) \).

When combining the concept of Lipschitz continuity with inclusion isotone interval extensions, we know that if a function \( F \) is an interval extension of \( f \) where \( f \) satisfies the standard Lipschitz condition, then it follows that the extension of \( f \) is a Lipschitz interval extension [28]. To this extent we have the following lemma
which states that if \( \mathcal{F} \) is a natural interval extension of \( f \) then \( \mathcal{F} \) is also Lipschitz continuous, with the same Lipschitz constant.

```isabelle
lemma natural_extension_transfer_lipschitz: 
  assumes positive_N: \( 0 < N \)
  and inc_isotonic_F: inclusion_isotonic \( F \)
  and interval_ext_F: \( F \) is natural interval extension of \( f \)
  and lipschitz_F: \( C\)-lipschitz on \( \text{set} \) \( (X, N) \) \( \mathcal{F} \)
  shows \( C\)-lipschitz on \( \text{set} \) \( (\text{uniform} \) \( \text{subdivision} \) \( X, N) \) \( \mathcal{F} \)
```

This requires the interval extension to satisfy our stronger definition of \( \text{is} \) natural interval extension discussed in Sect. 3.1. The weaker definition \( \text{is} \) interval extension cannot guarantee that the Lipschitz interval extension holds with the same Lipschitz constant for \( F \) that satisfies the standard Lipschitz condition. However, there does exist some Lipschitz constant \( L \) such that \( F \) is also Lipschitz continuous.

```isabelle
lemma lipschitz_on_implies_lipschitzI_on': 
  fixes \( f \) :: \( \text{real} \) interval
  and \( F \) :: \( \text{real} \) interval
  assumes non_empty: \( S \neq \emptyset \)
  and finite: finite \( S \)
  and non_zero_width: \( 0 < \text{Min} \) \( (\text{width} \ \text{'} \ S) \)
  and interval_ext_F: \( F \) is interval extension of \( f \)
  shows \( \exists L. L\)-lipschitz on \( S \) \( F \)
```

Among other important properties, we have proved that the composition of two Lipschitz continuous functions is also Lipschitz continuous in the context of interval arithmetic. Specifically if \( F \) and \( G \) are interval extensions with \( F \) Lipschitz in \( U \) and \( G \) Lipschitz in \( F(U) \). Then the composition \( H(X) = F(G(X)) \) is Lipschitz in \( U \).

```isabelle
lemma lipschitzI_on_compose: 
  fixes \( U \) :: \( \text{'} \text{a} \text{::}(\text{linordered_ring}) \) interval set
  and \( f \) :: \( C\)-lipschitz on \( U \) \( F \)
  and \( g \) :: \( D\)-lipschitz on \( (U \circ F) \) \( G \)
  shows \( (D \circ C)-lipschitz on \( U \) \( (G \circ F) \)
```

This allows us to reason about the composition of Lipschitz continuous functions, and is a direct extension of the property of Lipschitz continuous functions in real analysis to intervals.

### 3.4 Interval Division

The existing formalisation of the interval type in Isabelle (as part of the theory HOL-Library.Interval) does not provide a definition of division due to the fact that there is no universally accepted definition of division for intervals. See, for example, [34] for a discussion of various definitions of interval division. Still, most definitions (e.g., [27, 34]) satisfy the following two properties for intervals that do not include 0,

\[
\frac{1}{X} = \left[ \frac{1}{X}, \frac{1}{X} \right] \quad \text{for} \ 0 \not\in \text{set} \text{ of} (X) \quad (6)
\]

\[
\frac{X}{Y} = X \cdot \frac{1}{Y} \quad \text{for} \ 0 \not\in \text{set} \text{ of} (Y) \quad (7)
\]

Where \( X \) and \( X \) represent the lower and upper bounds of the interval \( X \) respectively. These two properties are sufficient to prove the main results of (extended) interval analysis (namely, convergence of Lipschitz continuous, inclusion isotone function as motivated in Sect. 2.2.1 and discussed in more detail in Sect. 3.7).

To capture the requirements of Equation 6 and Equation 7 we use Isabelle’s module concept, called locale [21]:

```isabelle
locale interval_division_inverse = inverse + 
  constrains inverse :: \( 'a \) interval \( \Rightarrow \) \( 'a \) interval
  and divide :: \( 'a \) interval \( \Rightarrow \) \( 'a \) interval
  assumes inverse_non_zero_def: 
    \( \forall i \in \mathbb{N}. i \Rightarrow (\text{inverse} \ ) x = \text{mk} \ _\text{interval} (1 \slash (\text{upper} \ ) x, 1 \slash (\text{lower} \ ) x) \)
  and divide_non_zero_def: 
    \( \forall i \in \mathbb{N}. i \Rightarrow (\text{divide} \ ) x \ y = x \times \text{inverse} \ y \)
begin . . . end
```

Subsequently, we can prove, once and for all, properties over all definitions of interval division that satisfy these two properties.

Isabelle defines (as Coq [2], Agda [37] or Lean [15]) division such that \( a/0 = 0 \) holds (the theorem \text{division} \_\text{ring} \_\text{divide} \_\text{zero} captures this fact in Isabelle/HOL). Thus, we can define integer division in a straight-forward manner for the type \text{real} interval:

```isabelle
definition inverse_interval where 
inverse_interval a = \text{mk} \ _\text{interval} (1 \slash (\text{upper} \ ) a, 1 \slash (\text{lower} \ ) a)

definition divide_interval where 
divide_interval a b = a \times \text{inverse} \ b
```

This definition satisfies the requirements of Equation 6 and Equation 7 and hence, it can provide an interpretation of the locale \text{interval} \_\text{division} \_\text{inverse}.

### 3.5 Extended Interval Arithmetic

More standard definitions of interval divisions (e.g., following the proposal of [27]) require the use extended reals, i.e., the extension of \( \mathbb{R} \) with the two infinity elements \(-\infty\) and \( +\infty\). A first attempt to define interval division over extended reals is:

```isabelle
definition inverse_interval where 
inverse_interval a = ( 
  if \( \neg \ 0 \in \text{set} \text{ of} \ a \) 
    then \text{mk} \ _\text{interval} (1 \slash (\text{upper} \ ) a, 1 \slash (\text{lower} \ ) a) 
    else if lower a = 0 \land \ 0 < upper a 
      then \text{mk} \ _\text{interval} (0, \text{upper} a, +\infty) 
      else if lower a < 0 \land \ 0 < upper a 
        then \text{mk} \ _\text{interval} (-\infty, \text{upper} a) 
        else \text{mk} \ _\text{interval} (-\infty, 1 \slash (\text{lower} \ ) a)
)

definition divide_interval where 
divide_interval a b = a \times \text{inverse} \ b
```

This definition also satisfies the requirements of Equation 6 and Equation 7. One drawback of this definition is that for the case \( lower \ a < 0 \land \ 0 < upper \ a \) the over-approximation \( [-\infty, +\infty] \) is very imprecise. Actually, Moore et al [27] propose to return the set \( \{\infty \ldots (1 \slash (\text{lower} \ ) a)\} \cup \{(1 \slash (\text{upper} d)\ldots\infty\} \).

Formalising this proposal in Isabelle requires to change the type of the division from \( \text{'a interval} \Rightarrow \text{'a interval} \to \text{'a interval} \Rightarrow \text{'a interval} \) list:
Thus, developing full support for extended interval arithmetic requires lifting all operations over intervals to multi-intervals (or lists of intervals).

### 3.6 Multi-Intervals

As motivated in the previous section, we need to model list of intervals, also called multi-intervals. In our Isabelle formalisation, we provide infrastructure to work with different types of multi-intervals, depending on whether overlapping or adjacent intervals should be preserved or automatically merged.

The most important multi-interval type in our formalisation represents non overlapping multi intervals, or adjacent intervals:

```isar
typedef (overloaded) 'a minterval_adj = 
  (is::'a interval list. valid_mInterval_adj is)
morphisms bounds_of_minterval_adj minterval_adj
unfolding valid_mInterval_adj_def
apply(intro exI[where x=[interval (1,1) ] for 1])
by(auto simp add: sorted_wrt_lower_def
non_overlapping_sorted_def)
```

Where valid_mInterval_adj is a predicate describing restricting the list of intervals to non-overlapping intervals that are sorted by their lower bound:

```isar
definition valid_minterval_adj :: 'a interval list => bool where
valid_minterval_adj is = distinct is ∧ is ≠ [] ∧ sorted_wrt (λ x y. upper x ≤ lower y) is
```

The type 'a minterval_adj provides a canonical representation of multi-intervals.

Furthermore, we define helpers to lift interval operations to list of intervals. For instance, we can define the addition over list of intervals as follows:

```isar
definition bin_op_interval_list where
bin_op_interval_list op xs ys = concat (map (λ x . map (op x) xs) ys)
```

Such definitions over list of intervals form the basis for the definitions for our multi-interval type 'a minterval_adj:

```isar
lift_definition plus_minterval_adj :: 'a minterval_adj ⇒ 'a minterval_adj ⇒ 'a minterval_adj
is λ a b . mk_mInterval_adj (list_plus a b)
```

We then prove various algebraic properties, for instance, that addition over adjacent multi-intervals is commutative, and instantiate type classes (respective interpret locales) to lift generic lemmas to multi-intervals.

### 3.7 Subdivisions and Refinements

The properties inclusion isotonicity and Lipschitz continuity can be combined to lead us to an important property of interval arithmetic. By splitting an interval \( X \) into \( N \) subdivisions and taking the union of the interval evaluations of the extended function \( F \) applied to each element in the subdivisions, we can compute arbitrarily sharp upper and lower bounds on the interval (recall Sect. 2.2.1). This is proven to converge linearly in the width of the subdivisions of the original interval \( X \).

However, in order to formally verify this convergence of refinement property in Isabelle/HOL, we first need to define the behaviour of some key components of the proof.

Firstly, in Sect. 2.2.1 we mention the term subintervals, this requires a method of dividing an interval \( X \) into \( N \) partitions. For the purpose of this work we define a version of partitioning \( X \) into \( N \) uniform subdivisions, all of equal width, and call this definition uniform_subdivisions. Mathematically a uniform subdivision can be described as the following:

\[
X = \left[ X + \frac{w(X)}{N}, X + \frac{w(X)}{N} + \frac{w(X)}{N} \right] \quad \text{for } i = 0, \ldots, N - 1
\]

Where \( w(X) \) represents the width of the interval \( X \) and \( N \) is the number of subdivisions. We model this in Isabelle as follows:

```isar
definition uniform_subdivision where
uniform_subdivision X N = map (λ i. let M = of_nat i in 
mk_interval (lower X + (upper X - lower X) * (i' + 1) / of_nat N)) 
[0..<N]
```

From this definition of uniform_subdivision it follows that the width of the interval \( X \) at subdivision \( i \) is \( w(X)/N \) where \( N \) is a natural number representing the number of subdivisions. Furthermore, the union of all the subdivisions results in the original interval \( X \), both of which are proven in Isabelle/HOL. In addition to these properties, we have also proven that the results obtained from uniform_subdivision meets our definition of a valid multi-interval type described in Sect. 3.6.

The increased granularity that uniform_subdivision provides over the interval \( X \) allows for a more detailed analysis over the behaviour of the function \( F \) over that interval. In this context, refinement of the function \( F \) over the interval \( X \) is defined as the union of the interval values of \( F \) applied over the all elements in the uniform subdivision. In other words, a refinement is a more precise representation of the function \( F \) over the interval \( X \). Mathematically this is defined as

\[
F_N(X) = \bigcup_{i=0}^{N} F(X_0, \ldots, X_{N-1})
\]

Where \( F_N(X) \) is the refinement of interval \( F(X) \), using the partition of \( X \) into \( N \) subdivisions.
In Isabelle/HOL we define refinement as the following, where interval_list_union takes the union of the list of intervals returned from uniform_subdivisions.

\[
\text{definition refinement where}
\]
\[
\text{refinement } F : X =
\]
\[
(\text{interval_list_union (map } F \text{ uniform_subdivision } X) \text{ N})
\]

From the fundamental theorem of interval analysis (see Equation 5) we know that \( f(X) \subseteq \text{set}(F(X)) \), and from the example in Sect. 2.2.1, we can clearly see that the extension \( F \) is an over approximation of \( f \). Hence, there is some excess interval \( E \) such that when this is added to the result of \( f \) applied onto every \( x_n \in (X_1 \ldots X) \) the result is \( F(X) \). We call this the excess width of \( F(X) \), and we have defined it as the following in Isabelle.

\[
\text{definition excess_width where}
\]
\[
\text{excess_width } F : X = \text{width}(F(X)) - \text{width_set}(f \ \text{set_of } X)
\]

The function \( f((X_1 \ldots X)) \) has an excess width of zero, it gives the exact range of values for \((X_1 \ldots X)\), whereas \( F(X) \) introduces an over approximation. However, we can compute \( f((X_1 \ldots X)) \) as closely as desired by the repeated subdivision and refinement of \( F(X) \). In fact, the excess width of a refinement is of order \( 1/N \) where \( N \) is the number of subdivisions.

Recall that we have some excess width \( (E) \) of an interval result of \( F(X) \), for each interval we then have \( F(X) = f((X_1 \ldots X)) + E \). This statement is also true for all of the subdivisions \( X_0 \) of \( X \), and so we can calculate the excess width of each subdivision \( X_0 \) to be \( w(E_0) \leq C \cdot w(X)/N \) where \( N \) is the number of subdivisions, \( X \) is the interval and \( X_0, E_0 \) are the intervals and excess width represented by a subdivision of \( X \). This has been proven in Isabelle with the following lemma:

\[
\text{lemma each_subdivision_excess_width_order:}
\]
\[
\text{fixes } X : \text{real interval}
\]
\[
\text{assumes inc_isotonic_F: inclusion_isotonic F}
\]
\[
\text{and lipschitz_f: lipschitz_f on } C U F
\]
\[
\text{and interval_ext_f: } F \text{ is interval_extension_of } f
\]
\[
\text{and lipschitz_f: Lipschitz_f on } \text{set (uniform_subdivision } X) \subseteq U
\]
\[
\text{and positive_N: } 0 < N
\]
\[
\text{and lipschitz_f: } L \text{-Lipschitz on } \text{set_of } X
\]
\[
\text{shows } X \subseteq \text{set (uniform_subdivision } X) \quad \text{N}
\]
\[
\text{excess_width } F : X \leq C \cdot \text{width } (X)/\text{of_nat } N
\]

This lemma holds due to the Lipschitz condition on the interval extension function \( F \). We know that the excess width of a subdivision of \( X \) is \( w(E_0) = w(F(X_0)) - w(f((X_1 \ldots X))) \), it follows that the excess width \( w(E_0) \leq w(F(X_0)) \). As \( F \) is a Lipschitz continuous, inclusion isotone extension of \( f \), the rate of change of the function \( F \) is bounded by our Lipschitz condition on intervals (Sect. 3.3), hence this means that \( w(F(X_0)) \leq C \cdot w(X_0)/N \), and therefore \( w(E_0) \leq C \cdot w(X_0)/N \), where \( N \) is the number of subdivisions.

When then looking at the excess width of the refinement of \( X \), (i.e., the union of all of the subdivisions after applying the function \( F \)), then the inequality holds with \( w(E) \leq w(F(X)) \leq 2 \cdot C \cdot w(X)/N \) as the upper limit of the excess width must be added to both the upper and lower bounds in the interval union. In Isabelle our lemma is the following (this is a formalisation of the paper proof in [28]):

\[
\text{lemma isotonic_lipschitz_refinement:}
\]
\[
\text{assumes positive_N: } 0 < N
\]
\[
\text{and inc_isotonic_F: inclusion_isotonic F}
\]
\[
\text{and interval_ext_F: } F \text{ is_interval_extension_of } f
\]
\[
\text{and lipschitz_f: } L \text{-Lipschitz_on } \text{set } X
\]
\[
\text{and lipschitz_F: Lipschitz_f on } \text{set } \text{uniform_subdivision } X
\]
\[
\text{shows excess_width_set F N f X}
\]
\[
\leq 2 \cdot C \cdot \text{width } X/\text{real } N
\]

Where excess_width_set and refinement_set are variants of excess_width and refinement that the interval result into their set equivalent.

As the result from uniform_subdivision is a contiguous list of intervals, when applying an inclusion isotone function \( F \) to this list, the result is a list of overlapping intervals. It is this overlapping that means only the excess width of the first and last intervals are considered in the union, and hence in the worst case scenario, the maximum excess width is to the left of the lower bound and to the right of the upper bound of \( X \), resulting in the inequality proven in isotonic_lipschitz_refinement.

This shows how we can compute arbitrarily close lower and upper bounds of a range of values represented as an interval \( X \) by subdividing the interval and taking the union of the function applied to all elements of the subdivision. This method converges with the width of the pieces in the subdivision, or as \( N \), the number of subdivisions, increases. It is important to note, that while in this paper, we focus on a uniform subdivision of the interval into \( N \) parts, the same result holds for a non-uniform, or weighted subdivision, with an adjustment of the rate of convergence. In this case, the method would converge with the width of the largest interval in the subdivision.

3.8 Discussion

A large part of the effort in formalising interval arithmetic was focused on straightening out the mathematical textbook notation that does not clearly distinguish between functions over real values vs. real intervals (and mostly assumes that for real operators there is an equivalent operation over real intervals). Furthermore, the textbook notation does not distinguish properly between intervals and their set representation—an aspect that needs to be clearly separated in a strongly typed systems such as higher-order logic. The need for lifting interval operations to multi-intervals is, essentially, neglected in the interval analysis literature (e.g., [28, 34]).

4 INTERVAL AWARE CLEAN

In this section, we provide a brief overview of the verification support we developed for Clean [36].

4.1 Polymorphism in Clean

We can define programs of interval types easily in Clean is due to the state architecture. Clean’s semantics and functionality are defined over a base control state, modelled in Isabelle/HOL as:

\[
\text{record control_state =}
\]
\[
\text{break_status : bool}
\]
\[
\text{return_status : bool}
\]
Records in Isabelle [30] are essentially named tuples where each field can have polymorphic typing. These records can also be extended with other named fields, importantly: if a record $B$ is an extension of $A$, then lemmas over $A$ also hold for $B$ (due to $B$ being a subtype of $A$).

Clean makes use of record extensibility when creating function states. For instance, Clean will automatically generate the `local_weighted_sum_state` for `weighted_sum` defined in Sect. 2.3, extending the `control_state` with the addition of the local variables `out :: 'a` and `i :: nat`:

```isar
record 'a local_weighted_sum_state = control_state
  out :: 'a
  i :: nat
```

Due to proof inheritance, Clean’s semantics and functionality proved on our weighted sum state. This makes working with types like intervals significantly easier than common deep embeddings like IMP2 [24], as we do not need to manually edit the core state formalisation.

Furthermore, the extensible state of Clean also allows us to define, similar to definition of functions in Sect. 3.1, that a Clean program $A$ is an interval extension of Clean program $B$. Informally, $A$ and $B$ differ only in their typing: in $B$ all types used in $A$ are lifted to their corresponding interval type. For instance, a variable of type `nat` in $A$ would be of type `interval nat` in $B$. This allows us to apply the concept of interval analysis to the analysis of programs.

In addition to the state, Clean’s semantics are defined over the `State Exception Monad`, which informally converts some state of type `σ` to some tuple of output and state or some exception:

```isar
type_synonym ('a, 'σ) MONEX = 'σ → ('a × 'σ)
```

Semantics being defined over some polymorphic output type allows for functions of any well-defined type to be parsed. Hence, functions over interval types do not require editing of the core command semantics.

## 5 CASE STUDY

In this section, we present a small case study, demonstrating how interval analysis can be combined with program verification in Isabelle/HOL. We build on the example shown in Sect. 2.3 and prove both the correctness and the inclusion isotonicity of performing a weighted sum over intervals. We also show that the interval weighted sum is an extension of the regular weighted sum. Being concise, only the interval correctness proof will be shown as the regular weighted sum has an analogous formalisation.

### 5.1 Formalisation in Clean

From Sect. 4.1, we know that integrating intervals to be a trivial exercise in Clean. Thus, requiring no semantic extension of vanilla Clean, our interval weighted sum is identical to the regular weighted sum from Sect. 2.3 with interval typing:

```isar
definition I_invariant xs ws ≡ (λσ. hd(out σ) = sum_list (map2 (+) (take (hd (i σ)) xs) (take (hd (i σ)) ws))) ∧ out σ ≠ [] ∧ i σ ≠ [] ∧ result_value σ ≠ [] ∧ ¬return_status σ)
```

### 5.2 Proof of Correctness

Our interval weighted sum has identical precondition and postconditions as in Sect. 2.3 and the correctness lemma is:

```isar
lemma I_weighted_sum_correct: Isabelle (Isar)
\[ λσ. σ' ⊲ I_weighted_sum_pre (xs, ws) σ' ∧ σ = σ ws \] 
I_weighted_sum (xs, ws) σ
\[ λσ. σ' ⊲ I_weighted_sum_post(xs, ws) σ' ∧ σ res \]
```

To prove this, unfolding the definition of `I_weighted_sum` leads to three sequential components:

1. `push_local_I_weighted_sum_state`: Initialises the local variables and result value to `[undefined]`.
2. `I_weighted_sum_core`: Contains the body of the Clean function
3. `pop_local_I_weighted_sum_state`: Pops the result value of the function and restores local variables.

We shall focus on item 2 as the others are trivial. When parsing each line, we will make use of our sequential composition WP rule:

\[
\begin{align*}
S_1(\sigma) \neq \text{None} & \quad \text{wp } (S_1 \dashv S_2) Q \; Q \; \sigma \\
\text{wp } S_2 & \quad (\text{wp } S_1 \; Q) \; \sigma \\
\end{align*}
\]

To use this, we require the termination of $S_1$ to simplify the WP of $S_1 \dashv S_2$. Parsing the assignment of `out := 0` and `i := 0` requires the local assignment simplification rule on normal execution states:

\[
\begin{align*}
\sigma & \quad \text{wp } f_{\text{update}} \quad (\text{wp } \text{hd} (\lambda \_ g \sigma)) \; \sigma \\
\end{align*}
\]

Where $f_{\text{update}}$ is a record specific update function for $f$ and `upd_hd` updates the value of $f$ with $g$. Using our adapted while loop and sequential rules, we require to show the correctness of a chosen invariant:

```isar
definition I_invariant xs ws ≡ (λσ. hd(out σ) = sum_list (map2 (+) (take (hd (i σ)) xs) (take (hd (i σ)) ws))) ∧ out σ ≠ [] ∧ i σ ≠ [] ∧ result_value σ ≠ [] ∧ ¬return_status σ)
```

---
Where the first line states that the value of out should be equal to the weighted sum of the input lists up to index \( i \). The conjunction out \( \sigma \neq [] \land i \sigma \neq [] \land \text{result\_value} \sigma \neq [] \) is trivial but required for Isabelle’s level of scrutiny. Finally, we require the loop to not change the return status of the control state.

We further require termination of the loop:

\[
\text{I\_ws\_measure \ x\ s \ w \equiv } \langle \lambda \sigma. \ \text{length} \ x - \text{hd}(i \sigma) \rangle
\]

Which is clearly decreasing for increasing \( 1 \sigma \). For more details about the formalisation, we refer readers to the artifact.

To close the correctness proof, we use our WP rule for returnRC. Since return\_result\_value (out) is assigning out to result\_value, the WP rule is essentially identical to the local assignment rule with the addition that return\_status is now True. This, combined with the post loop value of out, will show that our function truly implements an interval weighted sum.

5.3 Inclusion Isotonicity

Defining a more functional representation of our Clean function:

\[
\text{F} \ x \ y \ \sigma = \text{output} \ (\text{I\_weighted\_sum}(x, y)) \ \sigma
\]

We represent the inclusion isotonicity of our Clean function as:

\[
\text{lemma clean\_weighted\_sum\_inclusion\_isotonic:\n assumes } \sigma \neq \text{precond} \ \\
\text{and } \sigma \neq \text{postcond} \ \\
\text{shows } \langle \text{inclusion\_isotonic}\rangle (\text{length ws}) \langle \lambda x. \ F \ x \ x \ \sigma \rangle
\]

To prove this, we first will show that the Isabelle/HOL weighted sum is inclusion isotonic:

\[
\text{lemma weighted\_sum\_inclusion\_isotonic:\n fixes } X :: \text{real\ interval\ list} \\
\text{shows } \langle \text{inclusion\_isotonic}\rangle (\text{length ws}) \langle \lambda x. \ \text{weighted\_sum}(x, w) \ \sigma \rangle
\]

via the inclusion isotonicity of map2, foldr (+) and \( f \circ g \) for inclusion isotonic functions \( f \) and \( g \). Finally, using the correctness lemma, we can show the equivalence of \( F_{\text{ws}} \) to the Isabelle/HOL representation, which will close the goal of inclusion isotonicity.

5.4 Proof of Interval Extension

We now wish to show that our interval weighted sum over reals is an interval extension of the weighted sum over regular reals. By similar means to the previous section, we define an identical functional representation of the regular weighted sum:

\[
\text{definition } \text{f}_{\text{ws}} \ x \ y \ \sigma = \text{output} \ (\text{weighted\_sum}(x, y)) \ \sigma
\]

We now seek to show for every list \( X = [[x_1, x_1], \ldots, [x_n, x_n]] \) of degenerate intervals and weights \( W = [[w_1, w_1], \ldots, [w_n, w_n]] \):

\[
f_{\text{ws}} \ [x_1, \ldots, x_n] \ [w_1, \ldots, w_n] \ \sigma_{\text{ws}} = F_{\text{ws}} \ X \ W \ \sigma_{\text{ws}} \tag{10}
\]

Which in Isabelle, is formalised as:

\[
\text{lemma interval\_extension\_weighted\_sum:\n assumes } \text{precond} = \sigma_{\text{ws}} \land \text{weighted\_sum\_pre}(x, w) \ \sigma_{\text{ws}} \\
\text{and } \sigma_{\text{ws}} = \lambda x \cdot \text{weighted\_sum\_pre}(x, w) \ \sigma_{\text{ws}} \\
\text{and } \text{postcond} = \sigma_{\text{ws}} \land \text{interval\_of\_list}(x, w) \ \sigma_{\text{ws}} \\
\text{shows } \text{interval\_of}(f_{\text{ws}} x w \ \sigma_{\text{ws}}) = F_{\text{ws}} \ (\text{interval\_of\_list} x) \ (\text{interval\_of\_list} w) \ \sigma_{\text{ws}}
\]

To prove this, we first can use both correctness proofs to simplify the Clean implementation into standard Isabelle/HOL. Which is proven from our conclusion in Sect. 5.2 and the precondition assumptions. From this, the right-hand side is then \( \cap_{i=1}^{N} [x_i \cdot w_i, x_i \cdot w_i] \), which can be easily reduced to \( \cap_{i=1}^{N} [x_i, w_i, x_i \cdot w_i] \). By the linearity of interval w.r.t addition, it further simplifies to \( \cap_{i=1}^{N} x_i \cdot w_i, \cap_{i=1}^{N} x_i \cdot w_i \). This is precisely the L.H.S, closing the proof.

5.5 Fundamental Theorem of Interval Analysis

Given we have proven \( F_{\text{ws}} \) is both an interval extension of the regular weighted sum \( f_{\text{ws}} \) and inclusion isotonic, it should agree with the Fundamental Theorem of Interval Analysis. There is a difference, however, with how it has been formalised in Sect. 3.2 and our needs here. Our weighted sum is over lists of intervals, not plain intervals and so the lemma needs to be adjusted slightly:

\[
\text{lemma interval\_of\_list\_weighted\_sum:\n fixes } \ x :: \text{real\ interval\ list} \\
\text{assumes } \text{precond} = \sigma_{\text{ws}} \land \text{weighted\_sum\_pre}(x, w) \ \sigma_{\text{ws}} \\
\text{and } \sigma_{\text{ws}} = \lambda x \cdot \text{weighted\_sum\_pre}(x, w) \ \sigma_{\text{ws}} \\
\text{and } \text{postcond} = \sigma_{\text{ws}} \land \text{interval\_of\_list}(x, w) \ \sigma_{\text{ws}} \\
\text{shows } \text{interval\_of}(f_{\text{ws}} x w \ \sigma_{\text{ws}}) \subseteq \text{interval\_of\_list}(x) \ \sigma_{\text{ws}}
\]

The main modification is requiring each element of the list \( \text{interval\_of\_list} x \) to be a subset of \( X \). This is needed to prove the interval extension from Sect. 5.4. Using this, along with the inclusion isotonicity of \( F_{\text{ws}} \) we can close the proof.

5.6 Discussion

We now have a verified imperative program that uses interval types as first class citizen. This can be used for proving the correctness of numerical algorithms that rely on (extended) interval analysis. Furthermore, our program \( \text{I\_weighted\_sum} \) can also be understood as the (natural) interval extension of an imperative program \( \text{weighted\_sum} \) computing the weighted sum over reals. If, in the context of a larger program verification task, we need to show that the result of \( \text{weighted\_sum} \) stays within certain bounds, the proof can be reduced to symbolic evaluation.

6 RELATED WORK

There are a number of libraries offering a formalisation of interval arithmetic in interactive theorem provers. For example Coq\text{Interval}; this is a formalisation of interval arithmetic for the Coq proof assistant, developed mainly by Melquiond in [26] which provides tactics for proving inequalities on real-values expressions by using a combination of interval arithmetic and automatic differentiation.
A similar method has been implemented in PVS (Prototype Verification System) in [29] to efficiently approximate real valued functions in theorem provers. Most similarly to our presented work, Daumas et al [12] combine this method with their earlier work [14] to integrate interval splitting and Taylor series expansion to reduce the dependency effect of interval arithmetic in order to provide guaranteed proof of numerical properties. This work has been developed and verified in the theorem prover PVS.

There are also libraries developed in non-formal settings, for example Boost.Interval [6], a general C++ library that offers interval arithmetic, as well as Harapanahalli et al [17] who introduce a framework for interval analysis implemented directly in numpy.

Research on interval analysis and arithmetic is abundant, and can be used in many application areas, for example it is common to use interval analysis to mitigate rounding errors and solving differential equations (e.g. [16], [11] and [4]). However, early success has been seen when implementing the interval splitting and refinement method to approximate the output of a neural network. Wang et al use this method with an aircraft collision avoidance system (ACAS Xu [33]), to ensure that it adheres to strict security specifications [38]. Whereas Harapanahalli et al [17] use their framework to verify the output of network controlled systems.

In Isabelle/HOL in particular there are many cases of using interval arithmetic to prove the correctness of programs, for example Hölzl provides a proof method that translates inequalities into interval arithmetic calculations on floating-point numbers to compute boundary computations [18]. Boher uses this interval arithmetic to implement a programming language where all terms have 32-bit signed word values [3], and Immler and Hölzl use intervals and Lipschitz continuity to show a unique solution to multivariate case of the Picard-Lindelöf theorem [19], [20].

7 CONCLUSION AND FUTURE WORK
We presented a formalisation of interval analysis in Isabelle/HOL; in particular, we have proven formally the fundamental result that for inclusion isotone Lipschitz-continuous functions interval approximation converges to the value of the non-interval function. Furthermore, we extended this result to imperative programs. For the latter, we exploited the fact that Clean [36] has been designed for extensionality in mind.

We see two major lines of future work: First, our formalisation of interval arithmetic is reusable and can, for instance, be used for verifying the correctness of widely used numerical algorithms for analysing real world systems. Interesting targets include, for instance, the verification of the widely used Newton-Raphson method. Furthermore, interval analysis has been proven successful in the analysis of safety-critical neural networks, as pioneered by the Marabou [22]. Hence, combining our work with formal verification approaches of neural networks in Isabelle/HOL (e.g., [9]) would allow for the development of a neural network verification tool that is fully verified.

Second, the work on Clean shows the potential of the verification of programs over a polymorphic state. Hence, we plan to extend our weakest precondition calculus into a verification condition generator. Other applications areas that we consider of high importance are programs that integrate probabilistic variables (e.g., imperative programs that interact with neural networks) or that support ordinary differential equations as first class citizen (e.g., for the verification of cyber-physical systems).

AVAILABILITY.
The formalisation and case studies for Isabelle 2023, as described in this paper, are available to view on Zenodo [7]. The materials include both the Isabelle/HOL implementation and the detailed documentation generated by Isabelle. A maintained version that also will receive updates (including being ported to future versions of Isabelle) is available in the Archive of Formal Proofs (AFP) [8] under a BSD-license (SPDX-License-Identifier: BSD-3-Clause).

ACKNOWLEDGMENTS
This work was supported by the Engineering and Physical Sciences Research Council [grant number 670002170] and by Innovate UK as part of the project AutoCHERI.

REFERENCES


