UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Joosep Jääger

Implementation of affine arithmetic in Haskell

Bachelor's Thesis (9 ECTS)

Supervisor: Alisa Pankova, PhD
Supervisor: Dominique Unruh, PhD

Tartu 2020

Implementation of affine arithmetic in Haskell

Abstract:
Interval arithmetic and affine arithmetic are methods in numerical analysis that deal with ranges of numerical values. Affine arithmetic is often used instead of interval arithmetic since it can result in smaller errors. The result of this thesis is an affine arithmetic library written in Haskell. This library is written in a way that makes it more difficult to make errors when using it. The library was tested using certain mathematical properties of affine arithmetic.

Keywords:
Affine arithmetic, Haskell, interval arithmetic, numerical analysis

CERCS:
P170 Computer science, numerical analysis, systems, control

Afiinse Aritmeetika implementatsioon Haskellis

Lühikokkuvõte:
Intervalliaritmeetika ja afiinne aritmeetika on meetodid numbrilises analüüsis, mis võimaldavad läbi viia arvutusi arvuvahemikega. Kuna intervallarvutustega kaasnevad ebatäpsused, on võetud kasutusele afiinne aritmeetika, mis paljudel juhtudel annab täpsema vastuse. Töö käigus valmis Haskellis teek, mis võimaldab kasutada afiinset aritmeetikat teistes Haskelli programmides. Uus teek on loodud nii, et kasutajal on võimalikult keeruline selle kasutamisel vigu teha. Programmi korrektsuse tagamiseks kirjutati afiinse aritmeetika matemaatiliste omaduste põhjal testid.

Võtmesõnad:
Afiinne aritmeetika, Haskell, intervallaritmeetika, numbriline analüüs

CERCS:
P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Contents

1 Introduction **4**

2 Interval and affine arithmetic **5**
2.1 Operations in interval arithmetic 5
2.2 Limitations of interval arithmetic 6
2.3 Introduction to affine arithmetic 7
2.4 Affine operations 9
2.5 Non-affine operations 9
2.5.1 Chebyshev approximation 10
2.5.2 Min-range approximation 11
2.6 Representations of affine forms 12
2.7 Subdividing affine forms 13
2.8 Limitations of affine arithmetic 13
2.9 Handling edge cases 13
2.10 Applications of affine arithmetic 14

3 Implementation of affine arithmetic in Haskell **14**
3.1 Overview of Haskell 14
3.2 Implementation of the Hafar library 14
3.3 The AFM monad 15
3.4 Encapsulating the affine form 16
3.5 Testing with QuickCheck 17
3.6 Handling round-off errors 18
3.7 Using Hafar 19
3.8 Comparison with similar libraries 20

4 Conclusion **21**

References **23**
I. Licence 24

1 Introduction

Haskell is a purely functional programming language. It offers advantages when compared to other programming languages such as strong static typing, which helps the programmer avoid mistakes. However since Haskell is not as popular as some other languages, it can lack some more specific libraries.

Affine arithmetic is an improvement on interval arithmetic. Interval arithmetic is used to perform calculations on ranges of numerical quantities. Affine arithmetic has uses in many fields ranging from scientific calculations to ray-casting in computer graphics. There has been a surge of interest in using affine arithmetic to predict the behavior of electrical grids. The methods used in affine arithmetic were developed in the 90s and currently there are many implementations of affine arithmetic in C++ and various other languages. At the time of writing this thesis there are however only a few very limited implementations for Haskell. The result of this thesis is the *Hafar* affine arithmetic library for Haskell. This library aims to be easy to use and reasonably efficient and extensible.

Section 2 will give an overview of interval arithmetic and affine arithmetic. It will give the formal definitions and theorems that are necessary to reason about the implementation. This part should be understandable to anyone with some basic mathematical background.

Section 3 will describe the design and implementation of the *Hafar* affine arithmetic library. It will also give a brief overview of some of more advanced language features of Haskell that are used in the library. The last part of the thesis assumes knowledge of basic Haskell and functional programming in general.

2 Interval and affine arithmetic

Interval arithmetic (IA) is a model for numerical computation where each quantity is represented by an interval of numbers [SDF97, 15].

Let $\mathbb{I}$ denote the set of intervals where each interval is defined by its lower and upper bound e.g. $[-30, 5]$. Intervals can be thought to be representing an ideal value – usually some real number – between the upper and lower bound of that interval [Tup96, 23]. For example π can be represented by the interval $[3.14, 3.15]$.

2.1 Operations in interval arithmetic

The following notation is used by Tupper [Tup96, 23,24]. Let i^- and i^+ denote the lower bound and the upper bound of our interval respectively. Then that interval is denoted by the pair $[i^-, i^+]$ and the width of that interval is defined as

$$i^{||} \equiv i^+ - i^-.$$

The empty interval is denoted by $\emptyset$.

Every n-ary interval operation $g^{\mathbb{I}}$ which corresponds to the ideal operation $g^{\mathbb{R}}$ on real numbers has to satisfy the inclusion property:

$$\forall i \in \mathbb{I}^n \; \forall x \in i : g^{\mathbb{R}}(x) \in g^{\mathbb{I}}(i).$$

This means that any interval that we get as the result of applying the function to some tuple of intervals must contain the value that we get when applying the same operation to the ideal values that those intervals represent.

We will now look at some operations in interval arithmetic. These operations are defined by Stolfi and de Figueiredo [SDF97, 22–28].

Negation of an interval gives an interval, where both endpoints are negated and then switched around:

$$-[a, b] = [-b, -a].$$

Addition is done simply by adding the infima and suprema of the two intervals:

$$[a, b] + [c, d] = [a + c, b + d].$$

Subtraction is defined as adding the negation of the second interval to the first interval:

$$[a, b] - [c, d] = [a, b] + (-[c, d]) = [a - d, b - c].$$

Multiplication can be defined by multiplying the endpoints of the first interval with the endpoints of the second interval and then finding the minimum and maximum of those products.

$$[a, b] \cdot [c, d] = [\min A, \max A],$$

where

$$A = \{ac, ad, bc, bd\}.$$

This method of multiplying might not be the most efficient, since we only use two of the four calculated values. We can save some calculations by handling the multiplication case-by-case and calculate only the values that are necessary [SDF97, 26]:

```
def mul(x : 𝕀, y : 𝕀) : 𝕀
  if x = ∅ or y = ∅ then
    return ∅
  else if x = [0,0] or y = [0,0] then
    return [0,0]
  else if x^- ≥ 0 then
    if y^- ≥ 0 then
      return [x^- · y^-, x^+ · y^+]
    else if y^+ ≤ 0 then
      return [x^+ · y^-, x^- · y^+]
    else
      return [x^+ · y^+, x^- · y^-]
  else if x^+ ≤ 0 then
    if y^- ≥ 0 then
      return [x^- · y^+, x^+ · y^+]
    else if y^+ ≤ 0 then
      return [x^+ · y^+, x^- · y^-]
    else
      return [x^- · y^+, x^- · y^-]
  else
    if y^- ≥ 0 then
      return [x^- · y^+, x^+ · y^+]
    else if y^+ ≤ 0 then
      return [x^+ · y^-, x^- · y^-]
    else
      let a = min(x^- · y^+, x^+ · y^-)
      let b = min(x^- · y^-, x^+ · y^+)
      return [a, b]
```

2.2 Limitations of interval arithmetic

Interval arithmetic suffers from certain limitations. In longer computation chains IA tends to overestimate the error as the relative accuracy of the intervals may decrease at an exponential rate after each sequential application of a function [SDF97, 37]. This problem

can be mitigated somewhat by subdividing the intervals, doing the same calculations on those intervals and combining the results, but this reduces the error only in a linear fashion (see Figure 1). That is to say, in order to reduce the error by a factor of n the interval has to be partitioned into n sub-intervals.

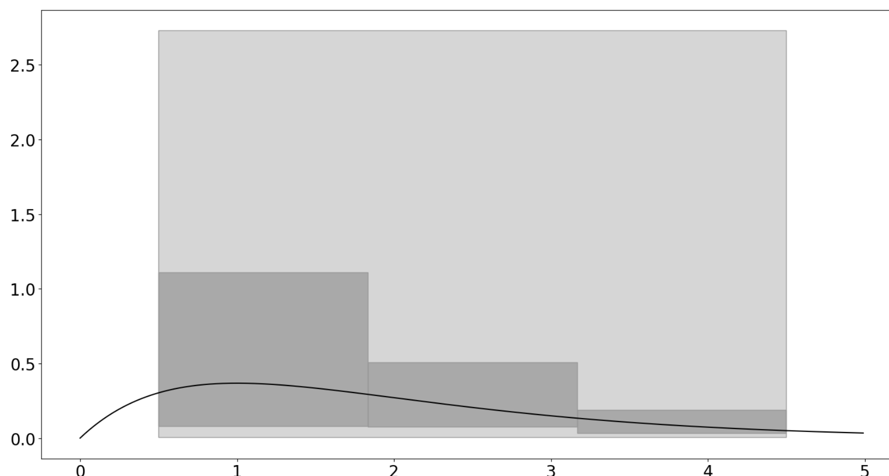

Figure 1. Subdivision of intervals on $f(x) = \frac{x}{e^x}$. Light gray area shows the result of evaluating with a single interval $[0.5, 4.5]$ and the dark gray boxes show the result of evaluating with three smaller subdivisions of that interval.

2.3 Introduction to affine arithmetic

Affine arithmetic (AA) is an improvement of interval arithmetic. Using this method, it is possible keep track of correlations between different quantities. The quantities in affine arithmetic are called affine forms. In this thesis we will use Rump and Kashiwagi's [RK15, 1102] definitions and notation. We define the set of all affine forms as $\mathcal{A} := \bigcup\{\mathcal{A}^k : k \in \mathbb{N}_0\}$, where $\mathcal{A}^k$ is a set of pairs $\{\langle c; \gamma\rangle : c \in \mathbb{R}, \gamma \in \mathbb{R}^k\}$.

An affine function ψ_C can be assigned to every affine form $C := \langle c, \gamma\rangle \in \mathcal{A}^k$ such that

$$\psi_C(\epsilon) := c + \sum_{i=1}^{k} \epsilon_i \gamma_i,$$

where $\epsilon \in U^k$ and $U = [-1, 1]$. When we apply this function to a tuple of values, it can be thought of as *fixing* the noise terms in order to get an ideal value.

The quantity c is called the midpoint of the affine form and the coefficients γ_i are called error terms. We can define the *range* of the affine form as:

$$range(C) := \left\{ c + \sum_{i=1}^{k} \gamma_i \epsilon_i : \epsilon_1, \ldots, \epsilon_k \in [-1, 1] \right\}.$$

This set can be thought of as the interval which corresponds to the affine form C.

Affine arithmetic is closely related to interval arithmetic. We can see that if a is an ideal value represented by $\langle c; \gamma \rangle$ then it is clear that the value is in the interval $[c - r, c + r]$, where $r = \sum_{i=1}^{k} |\gamma_i|$ [MMS15, 296]. For instance we can represent π with the affine form $\langle 3.1; 0.1 \rangle$. The quantity r is called the radius or half-width of the affine form [SDF97, 33–34].

It is possible to visualize the interdependence of two affine forms. For any affine forms C and D we can construct the set $\{(\psi_C(\epsilon), \psi_D(\epsilon)) : \epsilon \in U^k\}$, where k is the number of terms of the longer affine form. If the two affine forms are of a different length, we can use a *natural embedding* of that affine form, meaning that we pad the coefficients vector with enough zeroes to make the lengths match. When this set is plotted on a cartesian plane, it produces a *zonotope* (Figure 2) [RK15, 1102]. A similar set with intervals would produce a rectangle.

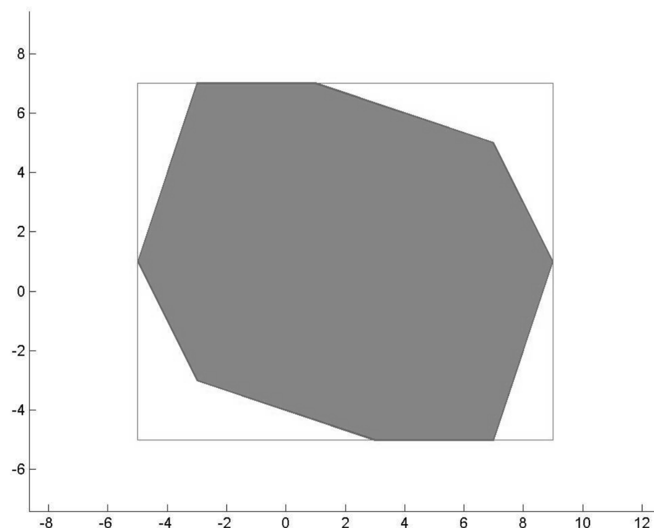

Figure 2. Zonotope produced by plotting two affine forms on a cartesian plane. The rectangle surrounding the zonotope shows the cartesian product of the ranges of these affine forms [RK15, 1102].

When we start defining operations in affine arithmetic we have to make sure that

these operations satisfy the fundamental invariant of affine arithmetic:

Proposition 1 (Fundamental invariant of affine arithmetic)**.** *[SDF97, 43–44] At any stable instant in an AA computation, there is a single assignment of values from U to each of the noise variables in use at the time that makes the value of every affine form equal to the value of the corresponding quantity in the ideal computation.*

This property will also be important later when testing our implementation of affine arithmetic with random tests.

2.4 Affine operations

An operation is an affine operation if it can be applied to affine functions ψ_C and ψ_D of affine forms C and D so that the result is the affine function of some affine form [SDF97, 50].

The following operations are defined in S. Rump and M. Kashiwagi's paper *Implementation and improvements of affine arithmetic* [RK15, 1102–1103].

Negation is an example of an affine operation. It is also one of the few exact operations on IEEE 735 floating point numbers, since negation is just a matter of flipping the sign bit. We can easily find the negation of an affine form $C := \langle c; \gamma \rangle$:

$$-C = \langle -c; -\gamma \rangle.$$

Addition can also be defined as an affine operation. Let $C := \langle c; \gamma \rangle$ and $D := \langle d; \delta \rangle$. Then

$$C + D = \langle c + d; \gamma + \delta \rangle.$$

Here we can think of γ and δ as vectors, so $\gamma + \delta$ is simply vector addition.

Subtraction can be defined through negation and addition:

$$C - D = C + (-D) = \langle c; \gamma \rangle + \langle -d; -\delta \rangle = \langle c - d; \gamma - \delta \rangle.$$

2.5 Non-affine operations

Some operations do not directly result in affine forms. In such operations we have to choose some affine function that approximates the function reasonably well. It must be guaranteed that the function is never underestimated, because that would contradict the fundamental invariant of affine arithmetic. There are many ways to measure how good an approximation is. We will take a look at two ways to come up with such approximations.

2.5.1 Chebyshev approximation

There are $n+1$ degrees of freedom in the choice of our affine approximation for $\gamma \in \mathbb{R}^n$. In general it would be more reasonable to limit ourselves to just having to pick a few coefficients. Let us only consider approximations to ψ_A that take the form

$$\psi_A(\epsilon) = \alpha\psi_C(\epsilon) + \beta\psi_D(\epsilon) + \zeta,$$

where C and D are the input affine forms of the function we are approximating and α, β, ζ are the coefficients that we need to find [SDF97, 54]. That limits our degrees of freedom to $n+1$ for any n-ary function.

Let us now look at the Chebyshev approximation, also known as minimax approximation, which aims to minimize the absolute error of the approximation [SDF97, 56]. The following definition is useful for formally reasoning about affine approximations:

Definition 1. *[RK15, 1103] We say that a triplet $[[p, q, \Delta]] \in \mathbb{R}^3$ represents $f : \mathcal{D} \subset \mathbb{R} \to \mathbb{R}$ on $I \in \mathbb{I}$, $I \subseteq \mathcal{D}$, if*

$$\forall x \in I : |px + q - f(x)| \leq \Delta.$$

Now we can find the triplet which corresponds to the Chebyshev approximation of our ideal function f:

Theorem 1. *[RK15, 1103] Suppose f is convex or concave on $I = [a, b]$ with $a \neq b$. Define*

$$p = \frac{f(b) - f(a)}{b - a}.$$

By mean-value theorem let $\zeta \in I$ such that $f'(\zeta) = p$. Define

$$q = \frac{f(a) + f(\zeta) - p(a + \zeta)}{2}, \;\; \Delta = \left|\frac{f(\zeta) - f(a) - p(\zeta - a)}{2}\right|.$$

Then $[[p, q, \Delta]]$ represents f on I.

We can use the triplet to define an approximation function F, which satisfies the following property:

Theorem 2. *[RK15, 1104] Let $C = \langle c; \gamma \rangle \in \mathcal{A}^k$ and let $[[p, q, \Delta]]$ represent $f(x) : \mathbb{R} \to \mathbb{R}$ on $range(C)$. Let $F : \mathcal{A}^k \to \mathcal{A}^{k+1}$ be defined as*

$$F(C) = \langle pc + q; p\gamma, \Delta \rangle.$$

Then

$$\forall \epsilon \in U^k \; \exists \epsilon' \in U : f(\psi_C(\epsilon)) = \psi_{F(C)}(\bar{\epsilon})$$

for $\bar{\epsilon} := (\epsilon, \epsilon')$.

Here the commas after vectors $p\gamma$ and ϵ are used to denote that the value is appended to the vector. Notice that the approximation function F returns an affine form with one extra error term Δ. This term has to be large enough so that whenever we apply the ideal function f to the ideal values of our affine forms, the resulting value must be within the range of our approximated affine form. This is in accordance with the fundamental invariant of affine arithmetic (Proposition 1).

2.5.2 Min-range approximation

Another way to approximate an affine operation is to use the min-range approximation. As the name hints, it minimizes the range of the resulting affine form.

Theorem 3. *[RK15, 1103] Let $I \in \mathbb{I}$ and twice differentiable $f : I \to \mathbb{R}$ be given. Suppose f is convex or concave on I and $f'(x) \neq 0$ on I. Let $p = f'(a)$ if $f'(x)f''(x) \geq 0$ and $p = f'(b)$ otherwise. Then $[[p, q, \Delta]]$ represents f on I with*

$$q = \frac{f(a) + f(b) - p(a+b)}{2}, \Delta = \left|\frac{f(b) - f(a) - p(b-a)}{2}\right|.$$

Figure 3 shows the difference of the minimax and min-range approximations on the function $exp(x)$.

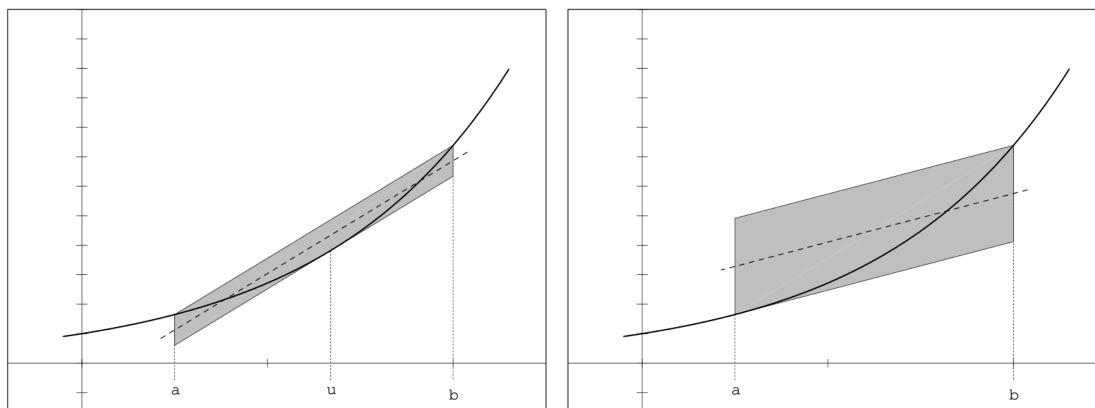

Figure 3. Chebyshev (left) and min-range (right) approximations of the exponential function [SDF97, 68].

Let us apply the min-range approach to estimate the reciprocal of an affine form. Let $C := \langle 2; 1 \rangle$, $range(C) = [1, 3]$. Our function is $f(x) = \frac{1}{x}$ and hence $f'(x) = -\frac{1}{x^2}$, $f''(x) = \frac{2}{x^3}$.

Next we will find the values p, q and Δ. Note that $f'(x)f''(x)$ is negative since $range(C)$ contains only positive values. Therefore, by Theorem 3:

$$p = -\frac{1}{3^2},\ q = \frac{\frac{1}{1} + \frac{1}{3} - (-0.11)(1+3)}{2},\ \Delta = \left|\frac{\frac{1}{1} - \frac{1}{3} - (-0.11)(3-1)}{2}\right|.$$

Our estimate for $\frac{1}{x}$ in $[1, 3]$ is

$$F(C) = \langle pc + q; p\gamma, \Delta\rangle = \langle \frac{2}{3}; -\frac{1}{9}, \frac{2}{9}\rangle$$

and the corresponding interval is

$$range(F(C)) = \left[\frac{1}{3}, 1\right].$$

The multiplication of two affine forms is a non-affine operation. Let $C = \langle c, \gamma\rangle$ and $D = \langle d, \delta\rangle$, then

$$C \cdot D = \langle cd; c\delta + d\gamma, ||\gamma||_1||\delta||_1\rangle,$$

where $||x||_1 := \sum_{i=1}^{n} x_i,\ x \in \mathbb{R}^n$ [RK15, 1104].

This follows naturally from multiplying the affine functions of the two affine forms. The terms which contain products of error coefficients are added together to become the error term of the new affine form.

2.6 Representations of affine forms

In computer memory it might not always be reasonable to represent the affine forms as they are represented in the mathematical definitions. Messine [Mes02] compared different representations of affine forms in *Extensions of Affine Arithmetic: Application to Unconstrained Global Optimization*. We will take a look at two of the representations discussed in that paper.

Let us first look at the standard affine form **AF**. Such affine forms are represented by:

$$C = \langle c; \gamma\rangle.$$

Because we need to add a new term each time after performing a non-affine operation, **AF** tends to grow unnecessarily large. This is not very practical when storing affine forms in computer memory.

One alternative to this is the *first affine form* **AF1**. First affine forms are represented by

$$C = \langle c; \gamma, \gamma_e\rangle,$$

where the ϵ_e term represents all the errors due to non-affine operations. The error term γ_e, which we will call the anonymous error term, must be positive and chosen in such a way that it does not contradict the fundamental invariant of affine arithmetic (1). Such representation allows us do an arbitrary number of compositions of non-affine operations while maintaining a fixed number of terms. In our implementation of affine arithmetic this anonymous error term also accounts for round-off errors.

2.7 Subdividing affine forms

One way to improve the quality of an enclosure is to divide the original range of the variables into smaller intervals and considering the union of those smaller intervals. It is possible to apply branch-and-bound algorithm to the affine form to minimize the error.

Moscato, et al. [MMS15, 302 – 303] describe a method which requires two functions: a `subdivide` function that splits an affine form into two smaller affine forms and a `combine` function, which combines two affine forms into a larger one. Given an expression and a starting interval, the algorithm finds the resulting affine form when applying the expression to the original range. Every iteration the algorithm subdivides the interval that has the smallest precision and applies the expression to each of the new subdivisions. The algorithm stops when it recurses to some specified depth or when all the intervals have a sufficiently high precision.

Subdivision was used to benchmark different representations of affine forms in Messine's paper [Mes02]. Currently subdivision is not implemented in the *Hafar* library but this may change in the future.

2.8 Limitations of affine arithmetic

In some cases affine arithmetic can have a disadvantage over interval arithmetic. For instance let $C := \langle 2; 1 \rangle$, then $C \cdot C = \langle 4; 4, 1 \rangle$ and $range(C \cdot C) = [-1, 9]$. However when calculating the same product in interval arithmetic, we get a much tighter range:

$$A := range(C) = [1, 3],\ A \cdot A = [1, 9].$$

A remedy to this problem is to use the mixed IA/AA model as described by Rump and Kashiwagi [RK15, 1105 – 1106]. This means we run the same calculations on both the affine forms and their corresponding intervals. We then can find the range by finding the intersection of the resulting interval and the range of the resulting affine form.

2.9 Handling edge cases

In implementing affine arithmetic, there arises a question about what to do when the programmer attempts to take the reciprocal of an affine form containing zero or the logarithm of a negative affine from. In such cases it is difficult to decide what the program should do. In the C++ *YalAA* [Kie12] library this problem has been solved using policy objects. These would be stored within the affine form objects and would specify whether to ignore any invalid operations or to throw an exception.

Currently *Hafar* throws an exception when the user attempts to apply a function to an affine form where some of the values in the ranges are not within the domain of that function. In the future it can be reasonable to implement error handling in a way similar to *YalAA*.

2.10 Applications of affine arithmetic

Affine arithmetic has found uses in many fields. In computer graphics it has been used to render implicit surfaces[KHK+09] and to find polygonal approximations of implicit curves[dCNPDFS14]. Affine arithmetic is useful for modeling power inputs of microgrids which utilize variable energy sources. [VPB19]

3 Implementation of affine arithmetic in Haskell

This section will give an overview of Haskell and the implementation of the *Hafar* library. The full source code for *Hafar* is available on the GitHub repository [J2̈0a]. The *Hafar* module with documentation is also available on Hackage [J2̈0b].

3.1 Overview of Haskell

Haskell is a pure language, which means that programs written in Haskell operate on immutable data and there are no variables like in most imperative languages. It also means no function call is dependent on the state of the program and the result of the function call only depends on the parameters given to the function [Has14]. This pure approach to programming has a couple of advantages when compared to impure languages, for example it is generally much easier to reason about code, since pure functions have no side effects.

Pure programming languages also benefit from lazy evaluation, which ensures that values only get calculated once they are actually needed [Wad95].

The following explanation of monads is based on Wadler [Wad95]. Since all the data flow in pure languages has to be expressed explicitly, there tends to be a lot of code that only deals with moving data from its point of creation to its point of use. In impure languages such logistics can usually be done using global variables or other features utilizing side effects. In order to allow side effects in a pure language, Haskell uses a concept from category theory called monads.

3.2 Implementation of the Hafar library

In *Hafar* an affine form is defined as the following data structure:

```
1 data AF s a = AF a [a] a
```

Here the structure takes two type parameters `s` and `a`. Parameter `a` defines the type of the coefficients and parameter `s` is used to encapsulate the affine form inside the AFM monad described in Sec 3.3. The AF constructor takes three parameters. The first parameter is the midpoint of the affine form, the list contains all the coefficients $x_1 \dots x_n$

and the last parameter is the anonymous error term coefficient x_{n+1} as used in the **AF1** representation.

3.3 The AFM monad

Monads are a concept from category theory. It can be useful to think of monads as composable computations. In Haskell monads are implemented as the `Monad` typeclass. Every monad must implement the `return` and `bind` functions. The `return x` function returns a monad which simply evaluates to `x`. Monads can be combined using the bind function `(>>=)`. For our implementation we are specifically interested in the `State` monad, which allows us to access a global state when composing computations with affine arithmetic.

Whenever we create a new noise symbol, we have to ensure that it does not correspond to any previous noise symbol. To make sure this does not happen, we use the `State` monad to keep track of the array index of the last noise symbol. Since every function in Haskell is pure, meaning its value only depends on the parameters, we need to pass the state to the function explicitly. However it might be cumbersome to constantly keep track of the state, so we could use a state monad instead. The `State` monad comes with the functions `get` and `put`. These functions allow us to access and modify the state within the monad. Both functions return a state monad which we can then bind with other state monads to create a more complex computation. When we evaluate the state monad, we only need to supply it with an initial state and from there on everything is handled inside the monad.

In order to simplify managing stacks of monads, Haskell uses the concept of monad transformers. A monad transformer is a monad with an extra type parameter which stands for the monad that the transformer is modifying. Monad transformers can be stacked to create new monads with more complex capabilities [OGS08].

With that in mind, we define a new monad transformer `AFMT` as follows:

```
1 data AFMT t m a = AFMT {runAFMT :: AFIndex -> m (a,
    AFIndex)}
2 type AFM t a = AFMT t Identity a
3 type AFIndex = Int
```

This monad is very similar to the `StateT` monad transformer defined in Haskell for `State` monad except that we add a type variable `t`. This variable will be explained more in-depth in Sec 3.4. To make the monad more convenient to use we also define the `AFM` type synonym for a non-transformer version of the monad. AFMT is an instance of `Monad` and `MonadTrans` classes. All the necessary functions are defined just as in the `StateT` monad transformer.

It's important to note that the creation of new affine forms could be done without a monad. However, then it would be necessary to pass the previous state to each following

calculation explicitly and this would be very inconvenient.

3.4 Encapsulating the affine form

We define functions `newEps` and `newFromInterval` for creating new affine forms with previously unused noise symbols. The function `newEps` returns a new affine form with midpoint at zero that has only the new previously unused error term coefficient set to one. The `newFromInterval` returns an affine form with a fresh error term that has been scaled and shifted to match the range of the interval parameter. The interval arithmetic functions prepended with `IA` are from Kmett's *intervals* library [Kme20].

```
newEps :: Num a => AFM t (AF t a)
newEps = do
  idx <- get
  put $ idx + 1
  return $ AF 0 (replicate idx 0 ++ [1]) 0

newFromInterval :: ( Eq a
                   , Fractional a
                   , ExplicitRounding a)
  => IA.Interval a
  -> AFM t (AF t a)
newFromInterval i = do
  eps <- newEps
  -- Scale the AF to match the width of i
  let mult = ((IA.width i) / 2) .* eps
  -- Shift the AF to the midpoint of i
  return $ (IA.midpoint i) .+ mult
```

Notice that the `AFM` and `AF` both share a type parameter `t`. To evaluate the `AFM` monad, we use `evalAFM`:

```
evalAFM :: forall a b. (forall t. AFM t b) -> b
evalAFM (AFMT x) = fst . runIdentity $ x 0
```

We have defined the type of the function so that the type parameter `t` becomes bound on the left side of the arrow. Now if the type `b` were to contain the type parameter that was bound to this parameter `t`, it would now become a free variable and this would cause the type checker to give an error.

This prevents us from evaluating the monad to an affine form:

```
evalAFM newEps -- TYPE ERROR
```

but we can still evaluate other types of values:

```
evalAFM $ interval =<< newEps -- OK
```

This makes sense, since the interval that we calculated from the affine form cannot be changed back into the affine form itself, therefore it does not interfere with the state. The `runIdentity` is defined in the Haskell identity monad implementation and simply evaluates the underlying `Identity` monad of the `AFMT` monad transformer.

This method has been used in the `ST` monad to prevent the programmer from mixing references from different threads [LJ95, 8–9]. Since the check is done by the type checker the method does not have to do any checks at runtime.

In order to simplify the implementation of new operations, we defined the `minrange` function. It takes three parameters: a function f of type `a -> a`, the derivative of f and a data structure `Curvature`, which specifies whether f is convex or concave. It then uses the min-range theorem (Theorem 3) to derive an approximation with the type `AF a -> AF a`.

3.5 Testing with QuickCheck

Many programmers are familiar with unit tests, where the tester writes down test cases by hand and then checks whether they give the expected result. In contrast, QuickCheck is a library for random testing of program properties. It works differently from unit testing in that rather than specifying the test cases by hand, the programmer writes down properties and data generators. QuickCheck then uses these data generators to generate a large number of test cases automatically.

In *Hafar* we use QuickCheck to check the soundness of operations on affine forms.

```
correctnessPropUnary :: ( Fractional a
                        , Ord a
                        , Show a
                        , ExplicitRounding a)
  => (AF s a -> AF s a)
  -> (a -> a)
  -> [a]
  -> AF s a
  -> Property
correctnessPropUnary f g e x = withMaxSuccess 5000 $
    counterexample str res
  where af = f x
        rhs = g (IA.midpoint $ fix x e)
        rhs_lo = g (IA.inf $ fix x e)
        rhs_hi = g (IA.sup $ fix x e)
        res = rhs `IA.member` interval af .&&.
```

```
16                     rhs_lo `IA.member` interval af .&&.
17                     rhs_hi `IA.member` interval af
18             str = "-- RESULTS --\n"
19                ++ "- LHS -\n"
20                ++ "AF:          " ++ (show af)              ++ "\n"
21                ++ "INTERVAL: " ++ (show $ interval af) ++ "\n"
22                ++ "- RHS -\n"
23                ++ "MID: " ++ (show rhs)     ++ "\n"
24                ++ "HI:  " ++ (show rhs_hi) ++ "\n"
25                ++ "LO:  " ++ (show rhs_lo) ++ "\n"
```

This test checks whether Theorem 2 holds for a given function. The function takes two functions, a list of values in the interval $[-1, 1]$ and an affine form as its parameters. The parameters `f` and `g` correspond to the functions $f : \mathbb{R} \to \mathbb{R}$ and $F : \mathcal{A}^k \to \mathcal{A}^{k+1}$ accordingly.

The `fix` function fixes all the noise symbols of an affine form to the values in the list, except for the error term. It outputs an interval that is centered at the midpoint of the affine form after fixing the noise symbols and has a half-width equal to the value of the anonymous error coefficient of the affine form.

According to the theorem, when we fix the noise symbols in the affine form `x` and then apply function `g` to the result, we should get a value `rhs` that is contained within `interval $ f x`.

3.6 Handling round-off errors

In order to ensure that our affine forms give correct answers, we need to be able to control the rounding of floating point values. There is no way to specify the rounding method in the Haskell Prelude module. There are some external modules that allow the user to control the rounding of floating point number, one of such modules being the *rounded* module. Such libraries are usually wrappers around a C library.

In *Hafar* we define the `ExplicitRounding` class, so that data structures that are instances of that class have to implement the function `eps :: a -> a`.

For `Int`, the `eps` function is simply defined as `const 0`, since all operations with integers are exact. For single precision floats, this function is defined like so:

```
1 instance ExplicitRounding Float where
2 eps 0 = eps $ 2e-36
3 eps x = encodeFloat 2 (snd $ decodeFloat x)
```

The `decodeFloat` separates the mantissa and exponent of a floating point number and returns them as a pair of integers. When we set the integer representing the mantissa to 2 and then encode the result as a floating point number, we get a relatively small number

that still has an effect when added to or subtracted from the original value x. This method does not work well with zero, so we define it to return the eps of an arbitrary small value instead. Choosing an optimal value for eps is outside the scope of this thesis.

The ExplicitRounding class offers some useful functions such as next and prev, which return the supremum and infimum of the set of values that the argument of those functions can represent. It also offers some helper functions for doing rounded operations on the data, e.g. +/ adds two values and rounds the result towards positive infinity while +\ rounds the result towards negative infinity.

In the implementation of affine arithmetic operations, the rounding functions are used to calculate the error due to round-off errors.

```
add :: (ExplicitRounding a, Num a, Ord a) => AF s a -> AF
    s a -> AF s a
(AF x xs xe) `add` (AF y ys ye) = addError af rnd
  where zs  = (uncurry (+)) <$> embed xs ys
        af  = AF (x + y) zs (xe +/ ye)
        rnd = sumup $ (uncurry (+/)) <$> embed (eps <$> xs
             ++ [x]) (eps <$> ys ++ [y])
```

Here we estimate the round-off error by adding (rounded up) the eps values of all the values used in the calculation of the sum. The embed function zips together two lists of numbers, padding the shorter list with zeroes, and the sumup function works just like sum but it rounds every addition towards positive infinity.

3.7 Using Hafar

Following is an example of using *Hafar* to calculate the difference of two affine forms:

```
import Numeric.AffineForm
import Numeric.Interval hiding (interval)

x1 = do
  a <- newFromInterval $ 4...6
  b <- newFromInterval $ 4...6
  return . interval $ a - b

evalAFM x1
```

Here we see the do notation being used to compose a computation. We create two new affine forms, both with range $[4, 6]$. Because we created the affine forms separately, they do not share any noise terms and therefore when we evaluate the range of the difference of a and b, we get the interval $[-2, 2]$. If we were to calculate the difference

`a - a` instead, the range would evaluate to $[0, 0]$. When using floating point numbers, the results will have a small margin to account for any round-off errors.

3.8 Comparison with similar libraries

At the time of writing this thesis, there were only a few implementations of affine arithmetic. One of those libraries is the Levitate library [Cla19]. Levitate implements basic arithmetic and controlled rounding of values. However affine forms in Levitate only support double precision floating point values. It does not offer encapsulation of affine forms, so it is possible to mix affine forms from different state threads.

Hafar library in comparison has support for many elementary functions and can guarantee that threads will not be mixed. *Hafar* can be found from the Hackage repository. As opposed to other affine arithmetic libraries, our implementation defined the affine form as a polymorphic type, which means that the affine forms can be based on other numerical systems such as rationals or fixed point numbers. This means that the affine forms can be used with other numeric types, not just floating point numbers.

4 Conclusion

This thesis has given a formal overview of interval and affine arithmetic. We have used the fundamental invariant of affine arithmetic to derive the affine operations and approximations to non-affine operations. In order to derive the non-affine operations we have looked at the Chebyshev and min-range approximations.

These ideas were then implemented in Haskell as the *Hafar* library. The resulting library allows the user to use affine arithmetic in their program without introducing impurity into their code. We have also used the Haskell type system to prevent the user from accidentally mixing the states of different affine computations.

Future research could look into extending the library with more elementary functions. Another improvement that could be made to the library is to somehow allow the user to choose a way to handle the edge cases described in Section 2.9. Because the affine form implementation is type-polymorphic, it would also be interesting to see the library being used with number systems other than floating point numbers.

References

[Cla19] Kevin Clancy. kevinclancy/levitate, Oct 2019.

[dCNPDFS14] Filipe de Carvalho Nascimento, Afonso Paiva, Luiz Henrique De Figueiredo, and Jorge Stolfi. Approximating implicit curves on plane and surface triangulations with affine arithmetic. *Computers & graphics*, 40:36–48, 2014.

[Has14] HaskellWiki. Functional programming — haskellwiki,, 2014. [Online; accessed 27-February-2020].

[J2̈0a] J. Jääger. Hafar. `https://github.com/Soupstraw/hafar`, 2020. [Online; accessed 09-March-2020].

[J2̈0b] J. Jääger. Hafar. `https://hackage.haskell.org/package/hafar`, 2020. [Online; accessed 09-March-2020].

[KHK+09] Aaron Knoll, Younis Hijazi, Andrew Kensler, Mathias Schott, Charles Hansen, and Hans Hagen. Fast ray tracing of arbitrary implicit surfaces with interval and affine arithmetic. In *Computer Graphics Forum*, volume 28, pages 26–40. Wiley Online Library, 2009.

[Kie12] Stefan Kiel. Yalaa: Yet another library for affine arithmetic. *Reliable Computing*, 16:114–129, 2012.

[Kme20] E. Kmett. Intervals. `https://hackage.haskell.org/package/intervals`, 2020. [Online; accessed 09-March-2020].

[LJ95] John Launchbury and Simon L Peyton Jones. State in haskell. *Lisp and symbolic computation*, 8(4):293–341, 1995.

[Mes02] Frédéric Messine. Extentions of affine arithmetic: Application to unconstrained global optimization. *J. UCS*, 8(11):992–1015, 2002.

[MMS15] Mariano M. Moscato, César A. Muñoz, and Andrew P. Smith. Affine arithmetic and applications to real-number proving. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving*, pages 294–309, Cham, 2015. Springer International Publishing.

[OGS08] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008.

[RK15] Siegfried M Rump and Masahide Kashiwagi. Implementation and improvements of affine arithmetic. *Nonlinear Theory and Its Applications, IEICE*, 6(3):341–359, 2015.

[SDF97] Jorge Stol and Luiz Henrique De Figueiredo. Self-validated numerical methods and applications. In *Monograph for 21st Brazilian Mathematics Colloquium, IMPA, Rio de Janeiro. Citeseer*, volume 5. Citeseer, 1997.

[Tup96] Jeffrey Allen Tupper. *Graphing equations with generalized interval arithmetic.* University of Toronto, 1996.

[VPB19] Alfredo Vaccaro, Marina Petrelli, and Alberto Berizzi. Robust optimization and affine arithmetic for microgrid scheduling under uncertainty. In *2019 IEEE International Conference on Environment and Electrical Engineering and 2019 IEEE Industrial and Commercial Power Systems Europe (EEEIC/I&CPS Europe)*, pages 1–6. IEEE, 2019.

[Wad95] Philip Wadler. Monads for functional programming. In *International School on Advanced Functional Programming*, pages 24–52. Springer, 1995.

I. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Joosep Jääger**,
(author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Implementation of Affine Arithmetic in Haskell,
   (title of thesis)

   supervised by Alisa Pankova and Dominique Unruh.
   (supervisor's name)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Joosep Jääger
5/3/2020