yalaa - Yet Another Library for Affine Arithmetic

Implementation Manual

Stefan Kiel

May 26, 2011

1. Introduction

.....
As a template library, `YalAA` is usable with arbitrary (arithmetic) base types. However, only a specialization for the IEEE754 double is currently provided and discussion in the following sections.

2. Preliminaries

2.1. Interval Arithmetic

2.2. Affine Arithmetic

3. Affine Approximations

We can not carry out a non affine operation or function directly on affine forms. However, it is possible to use an affine approximation and bound the (non-linear) approximation error with an error term. Methods for calculating affine approximations are outlined and discussed in [2] by de Figueiredo and Stolfi. Following them, we will also limit ourselves to affine approximations of the form

$$f^a(\epsilon_1, ..., \epsilon_n) = \alpha\hat{x} + \beta\hat{y} + \zeta$$

for the two input forms $\hat{x}, \hat{y}$. Two methods for deriving such approximations are discussed there: The Chebyshev optimal affine approximation and the min-range approximation. Both methods are only suitable for functions which are strictly convex or concave over the considered domain. The latter is implemented in `YalAA` for those functions. For functions not satisfying these condition a Taylor series expansion is used, as described in Sect. **??**.

4. Implementation

4.1. Important Classes

In this section the most important classes of `YalAA` are described. Concepts are described in Sect. 4.2.

AffineForm `AffineForm` combines the several policy classes and concepts to a working object-oriented arithmetic type. It offers the usual operators and the elementary functions (Tab. ??). As a template class the function can be widely customized. The declaration is as follows:

```
template<typename T, \
         template<typename> class ET, \
         template<typename, template<typename> class> class AC, \
         template<typename, template<typename> class, template<typename, template<typename> class> class, class> class AR, \
         template<typename, template<typename> class, template<typename, template<typename> class> class> class AP, \
         typename EP, \
         typename IV>
class AffineForm { ... };
```

The base type `T` is used for representing the partial deviations. `AffineForm` makes no assumptions about this type, but you have to provide a specialization of the `base_traits` template for a custom type. Further the interval type `IV` and `T` have to fit into one another. A specialization of `base_traits` is required for `IV` also. The template parameters `ET, AC, AR, AP` are also templates, but *their* template parameters are automatically determined through template template parameters. They model the concepts `ErrorTerm`, `AffineCombination`, `AffinePolicy` and `ErrorPolicy` described in Sect. 4.2.

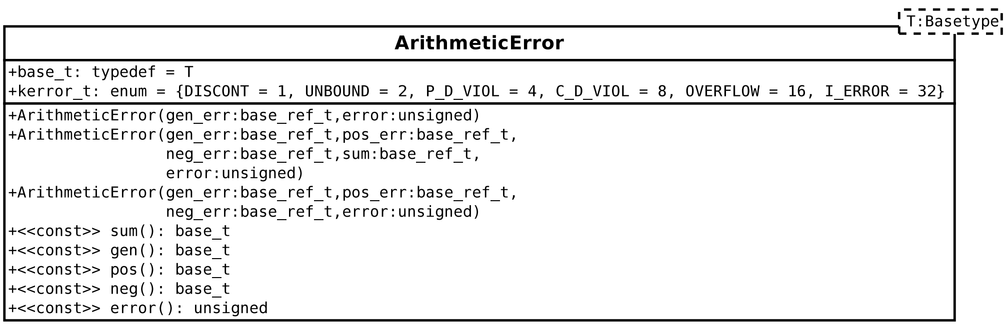

Figure 1: `ArithmeticError` class

ArithmeticError The `ArithmeticError` class is responsible for handling rounding errors and approximation errors. Further it stores information about computational errors *inside* the `ArithmeticKernel`. An overview of its operations is given in Fig. 1.

Rounding and approximations errors are split into three groups: *general*, *positive* and *negative*. General errors are identically to the standard error term model where ϵ_i is assumed to lie in the interval $[-1, 1]$. The latter two can combined with noise variables lying inside $[0, 1]$ and $[-1, 0]$ (cf. [4]). However, it's up to the concrete `ArithmeticKernel` whether it provides the extended model and up to the `AffinePolicy` how it maps the different error types. Obviously the both new terms can also combined with a standard ϵ_i. This enlarges the enclosures but nevertheless provides a valid bound. The errors are accessed through the `gen`, `pos`, `neg` functions. If only the sum is needed, it can retrieved with the `sum` function.

Further `ArithmeticError` defines the `kerror_t` type defining a minimum set of errors that every kernel should propagate:

`DISCONT` Function is discontinuous

`UNBOUND` Result has no finite bounds

`P_D_VIOL` Partial violation of natural domain

`C_D_VIOL` Complete violation of natural domain

`OVERFLOW` Overflow

`ERROR` Unknown error

The `DISCONT` error indicates that the operation is discontinuous over the affine form. An error of type `UNBOUNDED` is raised if the result of the operation is unbounded[1] By contrast `OVERFLOW` is used for indicating that the result is bounded but can not represented with used types. The `P_D_VIOL` indicates that a function was called with an argument lying partially outside its natural domain. For example let $\hat{x} = 0 + 2\epsilon_1$. Its range is $\boldsymbol{x} = [-2, 2]$. Then we can still evaluate $\sqrt{\hat{x}}$, but the negative parts of its range are ignored and `P_D_VIOL` is raised. If we modify our example and choose $\hat{x} = -2 + -1\epsilon_1$ with $\boldsymbol{x} = [-3, -1]$, $\sqrt{\hat{x}}$ will raise `C_D_VIOL` as the whole affine form lies outside of the square root's natural domain. or an overflow occurred. The last flag indicates a general error inside the kernel. Raised error flags can determined through the `error()` function, which returns an appropriate bitmask.

An `ArithmeticKernel` should support at least these flags. Otherwise error policies are not able to work correctly. It's up to the kernel to provide additional informations through extra flags. However, in this case a custom `ErrorPolicy` tightly coupled with the kernel is necessary for exploiting them.

4.2. Concepts

`YalAA` is a template library and can customized with policy classes[2]. It is possible to exchange policy classes in order to alter `YalAA`'s behavior. Policy classes do not have common base classes. Nonetheless they are sort of polymorphic, often called *static* or *compile-time* polymorphism. They achieve this by following a *concept*: An interface with a common set of `static` functions and *typedef*'s. In contrast to runtime-polymorphism there is currently no formal support for defining concept's in C++. So it's up to the programmer to follow the concepts or to face awful template related error messages ;-).

[1] As affine forms cannot represent improper intervals like $[x, \infty)$, the `UNBOUNDED` flag is raised in this case.

[2] See [?] for more information on policy-based design.

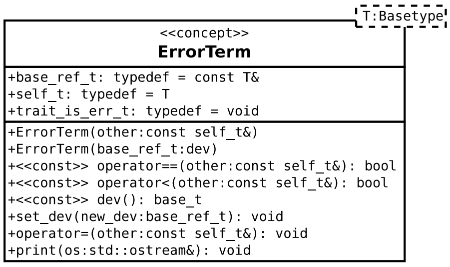

Figure 2: Concept defining an ErrorTerm

ErrorTerm An error term $\epsilon_i x_i$ is a combination of a symbolic noise variable ϵ_i and a partial deviation x_i. The partial deviation's type determines the base type of the affine form. While the concrete type of the ϵ's is implementation defined, it is necessary to define an ordering on them. If the constructor is called, the error term is responsible for acquiring a new previously unused noise symbol. The rest of the concept is straightforward, see Fig. 2 for a complete overview.

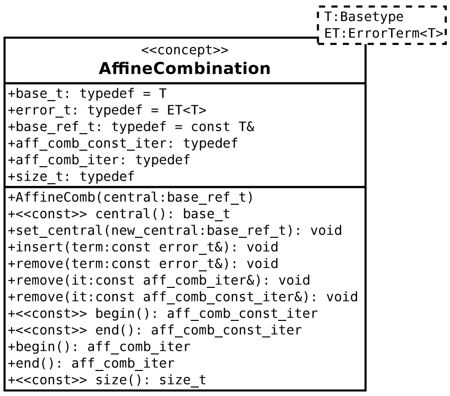

Figure 3: Concept defining an AffineCombination

AffineCombination This concept represents an affine combination $x_1\epsilon_i + \cdots + x_n\epsilon_n$ of error terms and a central value x_0 not associated with a symbolic noise variable. The iterators have to respect the ordering defined by the error terms. For the full concept cf. Fig. 3.

Affine operations Affine operations are not a template class but a bunch of free functions which can work on classes fulfilling the `AffineCombination` concept. A valid implementation has to provide functions with the following signatures:

```
template<typename AC>
unsigned mul_ac_s(AC* ac, typename AC::base_ref_t s);
```

```
template<typename AC>
unsigned add_ac_s(AC* ac, typename AC::base_ref_t s);

template<typename AC>
unsigned neg_ac(AC* ac);

template<typename AC, bool CENTRAL = true>
unsigned add_ac_ac(AC* ac1, const AC& ac2);

template<typename AC, bool CENTRAL = true>
unsigned sub_ac_ac(AC* ac1, const AC& ac2);
```

providing the usual affine operations. These consist of scaling (`mul_ac_s`) with a scalar of type `T::base_ref_t`, adding a scalar (`add_ac_s`) and negate an entire form. Further adding (`add_ac_ac`) and subtracting (`sub_ac_ac`) two affine combinations has to be supported. The result is always stored in the first affine combination supplied as argument.

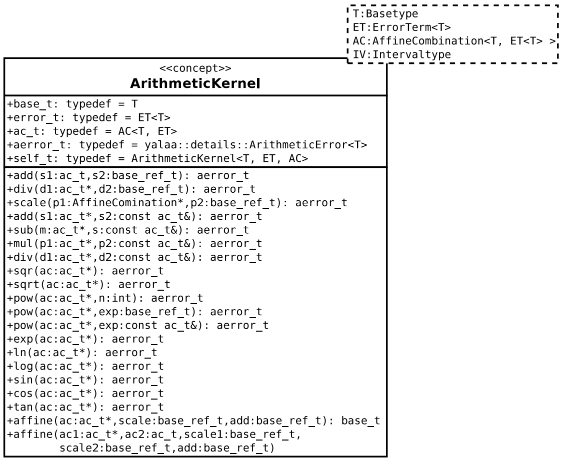

Figure 4: Concept defining an ArithmeticKernel

ArithmeticKernel The arithmetic kernel is the core of `YalAA` and carries out the actual mathematical operations. This design ensures easy interchangeability of operations. All operations are given in Fig. 4. All operations work on a given affine combination, which is altered during computation. However, they *must not* add any new error terms to the combination. Instead they deliver bounds of the occurring errors to the caller through an `ArithmeticError<T>` object. New noise terms are added through the `AffinePolicy`, which is described later on.

AffinePolicy The `AffinePolicy` controls the way new affine noise symbols are introduced. All operations are given in Fig. 5. The difference between the functions are

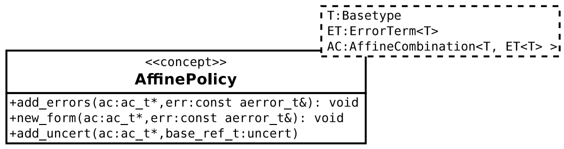

Figure 5: Concept defining an `AffinePolicy`

subtle. The `add_errors` function is called after performing an affine or non-affine combination. The function is responsible for adding the rounding and/or approximation error to the form. In contrast `new_form` is called if a new affine form is created, i.e. a new partially unknown quantity is introduced into the computation process. With `add_uncert` an uncertainty is introduced into the calculation process, this function is for example called if an affine form is combined with an interval quantity.

ErrorPolicy The `ErrorPolicy` is responsible for handling errors during the computation process. These errors are propagated through the error flags of the `ArithmeticError` class.

4.3. Supplied Implementations

`YalAA` is delivered with some standard implementations for the above described concepts.

ErrorTerm The supplied standard implementation `ErrorTermImpl<T>` uses an `unsigned long long`[3] and the `GCC`'s built-ins for atomic memory access. However, overflows in the ϵ's are not handled. You should replace the default implementation, if you need more than 2^{64} independent error terms.

AffineCombImpl The provided standard implementation `AffineCombImpl` uses a vector for storing the error terms. (TODO: Spezielle Formen wie AF1, AF2, GQF,...)

Affine operations A standard implementation is supplied in the file `affinecombopimpl.hpp`.

ArithmeticKernel `YalAA` contains a complete kernel for AA which can work with the IEEE 754 floating-point types `float`, `double`, `long double`. The implementation `ExactErrorFP` follows the approach described in [2] where possible. However, the there described approximation methods for non-affine operations are only applicable for strictly convex or concave functions. For other functions (e.g. $\cos, \sin, \tan$) `YalAA` uses a Chebyshev interpolation based approach as outlined in Sect. A.1. To ease reuse, we have split the `ExactErrorFP` kernel into several components (cf. Tab. 1).

[3]According to C99 at least 64-bits wide

Name	Operations	Remarks
ExactErrorAffineFPf	`add, sub, scale, neg`	Affine operations with exact floating-point error calculation, [2]
MinRangeBuiltInFP	`sqrt, inv`	Min-range, [2]
MinRangeFP	`exp, ln`	Min-range, needs IA library, cf. [2]
MultiplicationFP	`mul, sqr, pow`	Stolfi approach to multiplication, needs IA lib, [2], Sect. A.2
ChebyshevFP	`cos, sin, tan`	Chebyshev interpolation approach, needs IA lib, Sect. A.1

Table 1: Components of the `ExactErrorFP` kernel

AffinePolicy The policy class `AFO` supplied with `YalAA` mimics the behavior of the original affine arithmetic as described in [2]. All policy functions add the error with a new independent error symbol.

A. Function Implemention

This section briefly summarizes the concrete floating-point implementation for some functions provided by the `YalAA` built-in kernels.

A.1. Chebyshev Interpolation

The min-range approximation for calculating affine approximations to non-affine functions described in [2] is only applicable to differentiable strict convex or concave functions. As `YalAA` also supports elementary functions not satisfying these requirements, namely the trigonometric functions $\cos, \sin \tan$, another method is required. The standard arithmetic kernels supplied with `YalAA` use interpolation at the Chebyshev nodes for this. Readers interested in the mathematical details are referred to [1] and [3].

The Chebyshev nodes x_k are defined as

$$x_k = \cos\left(\frac{\pi(2k+1)}{2n+2}\right), k = 0...n$$

and are at the zeros of the Chebyshev polynomials

$$T_i(x) = \cos i\theta \text{ , if } x = \cos\theta.$$

A function $f : [-1, 1] \to \mathbb{R}$ can approximated using the n-th degree Chebyshev interpolant

$$p_n(x) = \frac{c_0}{2} + \sum_{k=1}^{n} c_k T_k(x)$$

with the Chebyshev coefficients

$$c_i = \frac{2}{n+1} \sum_{k=0}^{n} f(x_k) T_i(x_k).$$

For approximating a function on a general finite interval $[a, b]$ a linear transformation to $[-1, 1]$ is necessary. The new Chebyshev nodes are obtained through the inverse transformation as

$$x'_k = \frac{1}{2}\left((b-a)x_k + a + b\right)$$

and thus the coefficients as

$$c'_i = \frac{2}{n+1}\sum_{k=0}^{n} f(x'_k)T_i(x_k).$$

The new interpolant is

$$p_n(x) = \frac{c_0}{2} + \sum_{k=0}^{n} c'_k T_k(x)$$

for $x \in [-1, 1]$. We can transform any $x' \in [a, b]$ using a linear transformation $t : [a, b] \to [-1, 1]$ with

$$t(x') = \left(\frac{2x' - (a+b)}{b-a}\right).$$

The final polynomial for x' is thus

$$p_n(x') = \frac{c_0}{2} + \sum_{k=0}^{n} c'_k T_k(\left(\frac{2x' - (a+b)}{b-a}\right)).$$

Following [2] we are searching an affine operation of the form $\alpha\hat{x} + \zeta \pm \delta$ over the domain $\boldsymbol{x} = [a, b]$ of $\hat{x}$. This is a degree one polynomial, thus only the coefficients c'_0 and c'_1 are required. We calculate enclosures

$$\boldsymbol{\alpha} = \frac{2c'_1}{b-a}$$

and

$$\boldsymbol{\zeta} = \frac{c'_0}{2} - \frac{a+b}{b-a}$$

for α and ζ utilizing IA. The rounding error is shifted into the error term δ using

$$\delta = \frac{1}{2}\left((\operatorname{len}(\hat{x}) + 1)\operatorname{wid}\boldsymbol{\alpha} + \operatorname{wid}\boldsymbol{\zeta} + \operatorname{wid} R_1(\boldsymbol{x})\right)$$

where $\operatorname{len}(\hat{x})$ denotes the number of noise symbols in $\hat{x}$. Then we can use the midpoints $\operatorname{mid}\boldsymbol{\alpha}$ and $\operatorname{mid}\boldsymbol{\zeta}$ as α and ζ. A bound for $R_1(\boldsymbol{x})$ can derived with Lagrange remainder's formula[4]

$$R_1(\boldsymbol{x}) = \frac{(\operatorname{wid}\boldsymbol{x})^2 f^{(2)}(\boldsymbol{x})}{16}$$

[4]If a second derivative is available.

!!!! Ueberarbeiten, so wird es nicht gemacht !!!! The final result $\hat{x'}$ of the operation can determined by using the affine transformation

$$\begin{aligned} x_0' &= \alpha x_0 + \zeta \\ x_i' &= \alpha x_i' \end{aligned}$$

The occurring rounding errors are also shifted into δ, which is added as new error term $\delta\epsilon_k$, where ϵ_k is a new independent symbolic noise variable.

A.2. Integer Power-Function

The integer power function $\text{pow}(\hat{x}, n)$ for positive integers n. Using the binomial coefficients we can derive the formula

$$\begin{aligned} \hat{x}^n &= \left(x_0 + \sum_{i=1}^{m} x_i\epsilon_i \right) \\ &= x_0^n + n x_0^{n-1} \left(\sum_{i=1}^{m} x_i\epsilon_i \right) + \\ &\quad \binom{n}{2} x_0^{n-2} \left(\sum_{i=1}^{m} x_i\epsilon_i \right)^2 + \binom{n}{3} x_0^{n-3} \left(\sum_{i=1}^{m} x_i\epsilon_i \right)^3 + ... + \left(\sum_{i=1}^{m} x_i\epsilon_i \right)^n \end{aligned}$$

The first two terms of the sum are the affine part of the power function. Following the affine multiplication routine by de Figueiredo and Stolfi [2] the these terms are directly used for our affine approximation. The non linear terms are enclosed by a new error term. Let

$$\text{rad}\,\hat{x} = \sum_{i=1}^{m} |x_i|$$

so we get the error as

$$e = \binom{n}{k} (\text{rad}\,\hat{x})^k.$$

We split the error into an unsigned part e^+ for all terms with an even k and a signed $e^\pm$ one for all odd terms. The final error

$$e = \frac{1}{2} e^+ + e^\pm$$

is then added with a new noise symbol to the result. Finally the central value is adjusted by $\frac{1}{2}e^+$.

References

[1] S. Ackleh, E. J Allen, R. B. Kearfott, and Seshaiyer P. *Classical and Modern Numerical Analysis.* CRC Press, 2010.

[2] L.H. de Figueiredo and J. Stolfi. *Self-Validated Numerical Methods and Applications.* IMPA, Rio de Janeiro, 1997.

[3] J. C Mason and D. C. Handscomb. *Chebyshev Polynomials*. CRC Press, 2003.

[4] F. Messine. Extentions of affine arithmetic: Application to unconstrained global optimization. *Journal of Universal Computer Science*, 8(11):992–1015, 2002.