`real_asymp`: Semi-Automatic Real Asymptotics in Isabelle/HOL

Manuel Eberl

24 October 2026

Contents

1 Introduction **2**

2 Supported Operations **2**

3 Proving Limits and Asymptotic Properties **3**

4 Diagnostic Commands **5**

5 Extensibility **6**
5.1 Basic fact collections 6
5.2 Expanding Custom Operations 6
5.3 Extending the Sign Oracle 7

1 Introduction

This document describes the `real_asymp` package that provides a number of tools related to the asymptotics of real-valued functions. These tools are:

- The *real-asymp* method to prove limits, statements involving Landau symbols ('Big-O' etc.), and asymptotic equivalence ($\sim$)
- The **real-limit** command to compute the limit of a real-valued function at a certain point
- The **real-expansion** command to compute the asymptotic expansion of a real-valued function at a certain point

These three tools will be described in the following sections.

2 Supported Operations

The `real_asymp` package fully supports the following operations:

- Basic arithmetic (addition, subtraction, multiplication, division)
- Powers with constant natural exponent
- *exp*, *ln*, *log*, *sqrt*, *root k* (for a constant k)
- *sin*, *cos*, *tan* at finite points
- *sinh*, *cosh*, *tanh*
- *min*, *max*, *abs*

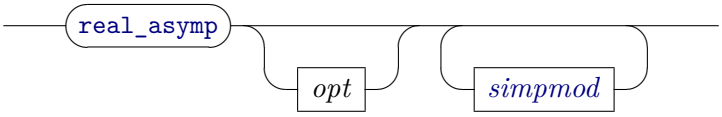
Additionally, the following operations are supported in a 'best effort' fashion using asymptotic upper/lower bounds:

- Powers with variable natural exponent
- *sin* and *cos* at $\pm\infty$
- *floor*, *ceiling*, *frac*, and *mod*

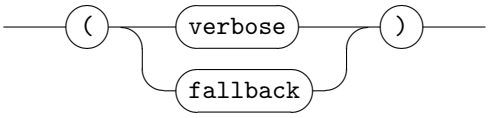
Additionally, the *arctan* function is partially supported. The method may fail when the argument to *arctan* contains functions of different order of growth.

3 Proving Limits and Asymptotic Properties

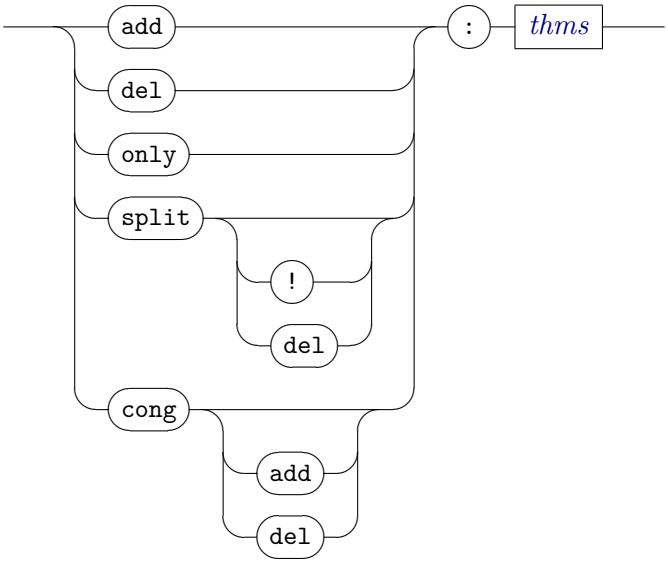
real-asymp : *method*



opt



simpmod



The *real-asymp* method is a semi-automatic proof method for proving certain statements related to the asymptotic behaviour of real-valued functions. In the following, let f and g be functions of type $real \Rightarrow real$ and F and G real filters.

The functions f and g can be built from the operations mentioned before and may contain free variables. The filters F and G can be either $\pm\infty$ or a finite point in $\mathbb{R}$, possibly with approach from the left or from the right.

The class of statements that is supported by *real-asymp* then consists of:

- Limits, i. e. *filterlim* f F G
- Landau symbols, i. e. $f \in O[F](g)$ and analogously for o, Ω, ω, Θ

- Asymptotic equivalence, i. e. $f \sim[F]\ g$
- Asymptotic inequalities, i. e. $\forall_F\ x\ in\ F.\ f\ x \leq g\ x$

For typical problems arising in practice that do not contain free variables, *real-asymp* tends to succeed fully automatically within fractions of seconds, e. g.:

lemma ‹*filterlim* (λ*x*::*real*. (1 + 1 / *x*) *powr x*) (*nhds* (*exp* 1)) *at-top*›
by *real-asymp*

What makes the method *semi-automatic* is the fact that it has to internally determine the sign of real-valued constants and identical zeroness of real-valued functions, and while the internal heuristics for this work very well most of the time, there are situations where the method fails to determine the sign of a constant, in which case the user needs to go back and supply more information about the sign of that constant in order to get a result.

A simple example is the following:

lemma ‹*filterlim* (λ*x*::*real*. *exp* (*a* ∗ *x*)) *at-top at-top*›
oops

Here, *real-asymp* outputs an error message stating that it could not determine the sign of the free variable *a*. In this case, the user must add the assumption $a > 0$ and give it to *real-asymp*.

lemma
assumes ‹*a* > 0›
shows ‹*filterlim* (λ*x*::*real*. *exp* (*a* ∗ *x*)) *at-top at-top*›
using *assms* **by** *real-asymp*

Additional modifications to the simpset that is used for determining signs can also be made with *simp add*: modifiers etc. in the same way as when using the *simp* method directly.

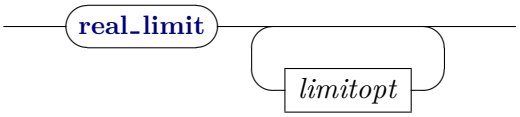
The same situation can also occur without free variables if the constant in question is a complicated expression that the simplifier does not know enough ebout, e. g. *pi* − *exp* 1.

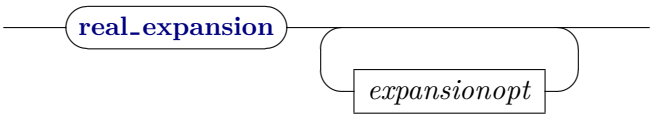
In order to trace problems with sign determination, the (*verbose*) option can be passed to *real-asymp*. It will then print a detailed error message whenever it encounters a sign determination problem that it cannot solve.

The (*fallback*) flag causes the method not to use asymptotic interval arithmetic, but only the much simpler core mechanism of computing a single Multiseries expansion for the input function. There should normally be no need to use this flag; however, the core part of the code has been tested much more rigorously than the asymptotic interval part. In case there is some implementation problem with it for a problem that can be solved without it, the (*fallback*) option can be used until that problem is resolved.

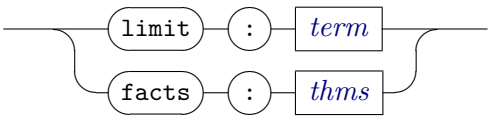
4 Diagnostic Commands

real-limit : *context* →
real-expansion : *context* →

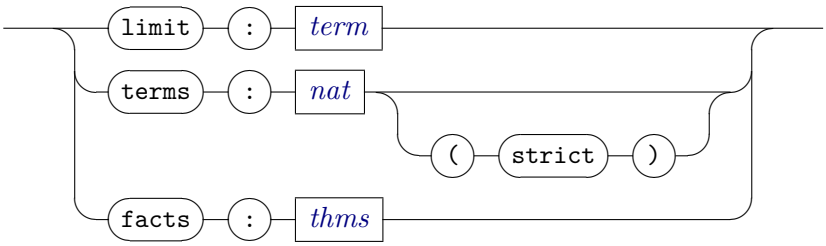




limitopt



expansionopt



real-limit computes the limit of the given function $f(x)$ for as x tends to the specified limit point. Additional facts can be provided with the *facts* option, similarly to the **using** command with *real-asymp*. The limit point given by the *limit* option must be one of the filters *at-top*, *at-bot*, *at-left*, or *at-right*. If no limit point is given, *at-top* is used by default.

The output of **real-limit** can be ∞, $-\infty$, $\pm\infty$, c (for some real constant c), 0^+, or 0^-. The $+$ and $-$ in the last case indicate whether the approach is from above or from below (corresponding to *at-right* 0 and *at-left* 0); for technical reasons, this information is currently not displayed if the limit is not 0.

If the given function does not tend to a definite limit (e. g. *sin* x for $x \to \infty$), the command might nevertheless succeed to compute an asymptotic upper and/or lower bound and display the limits of these bounds instead.

real-expansion works similarly to **real-limit**, but displays the first few terms of the asymptotic multiseries expansion of the given function at the given limit point and the order of growth of the remainder term.

In addition to the options already explained for the **real-limit** command, it takes an additional option *terms* that controls how many of the leading terms of the expansion are printed. If the (*strict*) modifier is passed to the *terms option*, terms whose coefficient is 0 are dropped from the output and do not count to the number of terms to be output.

By default, the first three terms are output and the *strict* option is disabled.

Note that these two commands are intended for diagnostic use only. While the central part of their implementation – finding a multiseries expansion and reading off the limit – are the same as in the *real-asymp* method and therefore trustworthy, there is a small amount of unverified code involved in pre-processing and printing (e. g. for reducing all the different options for the *limit* option to the *at-top* case).

5 Extensibility

5.1 Basic fact collections

The easiest way to add support for additional operations is to add corresponding facts to one of the following fact collections. However, this only works for particularly simple cases.

real-asymp-reify-simps specifies a list of (unconditional) equations that are unfolded as a first step of *real-asymp* and the related commands. This can be used to add support for operations that can be represented easily by other operations that are already supported, such as *sinh*, which is equal to $\lambda x.\ (exp\ x - exp\ (-\ x))\ /\ (2::'a)$.

real-asymp-nat-reify and *real-asymp-int-reify* is used to convert operations on natural numbers or integers to operations on real numbers. This enables *real-asymp* to also work on functions that return a natural number or an integer.

5.2 Expanding Custom Operations

Support for some non-trivial new operation $f(x_1, \ldots, x_n)$ can be added dynamically at any time, but it involves writing ML code and involves a

significant amount of effort, especially when the function has essential singularities.

The first step is to write an ML function that takes as arguments

- the expansion context
- the term t to expand (which will be of the form $f(g_1(x), \ldots, g_n(x))$)
- a list of n theorems of the form $(g_i \textit{ expands-to } G_i)\ bs$
- the current basis bs and returns a theorem of the form $(t \textit{ expands-to } F)\ bs'$ and a new basis bs' which must be a superset of the original basis.

This function must then be registered as a handler for the operation by proving a vacuous lemma of the form *REAL-ASYMP-CUSTOM f* (which is only used for tagging) and passing that lemma and the expansion function to `Exp_Log_Expression.register_custom_from_thm` in a **local-setup** invocation.

If the expansion produced by the handler function contains new definitions, corresponding evaluation equations must be added to the fact collection *real-asymp-eval-eqs*. These equations must have the form $h\ p_1\ \ldots\ p_m = rhs$ where h must be a constant of arity m (i. e. the function on the left-hand side must be fully applied) and the p_i can be patterns involving *constructors*.

New constructors for this pattern matching can be registered by adding a theorem of the form *REAL-ASYMP-EVAL-CONSTRUCTOR c* to the fact collection *exp-log-eval-constructor*, but this should be quite rare in practice.

Note that there is currently no way to add support for custom operations in connection with 'oscillating' terms. The above mechanism only works if all arguments of the new operation have an exact multiseries expansion.

5.3 Extending the Sign Oracle

By default, the `real_asymp` package uses the simplifier with the given simpset and facts in order to determine the sign of real constants. This is done by invoking the simplifier on goals like $c = 0$, $c \neq 0$, $c > 0$, or $c < 0$ or some subset thereof, depending on the context.

If the simplifier cannot prove any of them, the entire method (or command) invocation will fail. It is, however, possible to dynamically add additional sign oracles that will be tried; the most obvious candidate for an oracle that one may want to add or remove dynamically are approximation-based tactics.

Adding such a tactic can be done by calling `Multiseries_Expansion.register_sign_oracle`. Note that if the tactic cannot prove a goal, it should fail as fast as possible.