T.C.A.S.: **T**CAS **C**an **A**lways **S**olve

Numerical Analysis: a Concrete Computational Approach to Limit Evaluation

Roberto Alessandro Bertolini

Liceo Nervi Ferrari - Morbegno

Abstract

Evaluating limits by hand can become a trivial task with a bit of exercise, but a normal computer is generally incapable of proceeding intuitively and needs a reliable algorithm in order to be able to reach consistently the same result. While some purely heuristic or naive approaches might, at first glance, seem good enough, they tend to quickly fall apart in real world conditions. TCAS is a portable universal C implementation of the Gruntz Algorithm [3], which at the present day is the most efficient and reliable way of evaluating limits in the exp-log field of operations.

Contents

1 Introduction

1.1 The Limit of a Function

The limit of the function $f : \mathbf{R} \to \mathbf{R}$ is defined as the following:

Definition.

$$\lim_{x \to x_0} f(x) = l$$

If and only if $\forall \varepsilon > 0 \ \exists \ \delta(\varepsilon) \mid \forall x \in D_f, 0 <\mid x - x_0 \mid < \delta \implies \mid f(x) - l \mid < \varepsilon$, where $x_0, \varepsilon \in \mathbf{R}$

1.2 Polish Notation

Jan Lukasiewicz devised the so-called Polish Notation [11] in 1924; it is a prefix notation, where the operator precedes its operands. As long as the number of operands is predefined, there can't be an ambiguity in the order of evaluation, so this notation doesn't strictly require parenthesis. Consider the following expression written without parenthesis:

$$8 \times 4 + 3 \times 2 - 6$$

Depending on where the parenthesis are placed, it can evaluate to different results:

$$(8 \times 4 + 3) \times (2 - 6) = -140; \quad (8 \times (4 + 3) \times 2) - 6 = 106$$

Now consider a similar expression written using polish notation:

$$\times \times 8 + 4 \ 3 - 2 \ 6 = -224$$

It necessary evaluates to a single result. This makes parsing the expression into an abstract syntax tree [7] much easier, as the parser doesn't have to make educated choices about its interpretation.

1.3 The Shortcomings of the Naive Approach

Evaluating a limit might seem easy for a computer, as it should be able to continuously approximate the result with a smaller ε until the rounding error is acceptable enough, but can this always work? Consider the following function:

$$f : y = \frac{1}{x^{\ln \ln \ln \frac{1}{x} - 1}} \tag{1}$$

Its graph is the following: fig. 1, where it seems that for values of x closer to zero the function tends to zero, yet if we compute the limit: $\lim x \to 0^+ f(x) = +\infty$, so the function should have a vertical asymptote in zero. We just can't see it in the graph because it becomes visible for

$x \approx 4.29 \times 10^{-1656521}$. A computer trying to approximate the result would have stopped long before this value, returning zero.

What if we try instead to recursively apply L'Hôpital's rule [8] until we reach a clear result? Consider the following functions:

$$f(x) = e^x + e^{-x}$$

$$g(x) = e^x - e^{-x}$$

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f''(x)}{g''(x)} = \lim_{x\to\infty} \frac{e^x + e^{-x}}{e^x - e^{-x}}$$

So recursively applying L'Hôpital's rule would not work in this case.

2 The Gruntz Algorithm

2.1 Manipulating the Most Rapidly Varying Subexpressions

Definition. $g(x)$ is said to be a subexpression of $f(x)$ if the evaluation of $f(x)$ causes the evaluation of $g(x)$. This is represented with the following notation:

$$g(x) \lhd f(x), \text{ if } g(x) \text{ is a subexpression of } f(x)$$
$$g(x) \not\lhd f(x), \text{ if } g(x) \text{ is not a subexpression of } f(x)$$

Definition. Two subexpressions $g(x)$ and $h(x)$ can be compared based on their comparability class, which describes how rapidly they vary.

$$\begin{aligned} f(x) \prec g(x) \text{ if and only if } \lim_{x\to\infty} \frac{\ln |\, f(x)\,|}{\ln |\, g(x)\,|} = 0 \\ f(x) \succ g(x) \text{ if and only if } \lim_{x\to\infty} \frac{\ln |\, f(x)\,|}{\ln |\, g(x)\,|} \neq 0 \in \mathbf{R} \end{aligned} \tag{2}$$

Definition. The MRV set is the set that contains all the subexpressions of the highest comparability class.

$$mrv(f(x)) = \begin{cases} \{\} & \text{if } x \not\lhd f(x) \\ \{g(x) \mid g(x) \lhd f(x) \wedge (\nexists\ h(x) \lhd f(x) \mid h(x) \succ g(x)))\} \end{cases}$$

The first step of the Gruntz algorithm is recursively computing the MRV set of the limit that needs evaluating. If empty, the limit is indipendent of x. Once found, all the subexpressions contained in it must be rewritten as a function of w, which itself has to respect the following

expression:

$$\lim_{x\to\infty} w(x) = 0+$$

The most common w is $w(x) = e^{-x}$.

2.2 Power Series Representation

Definition. The Maclaurin series of any function is defined as the following:

$$\sum_{n=0}^{\infty} \frac{f^{(n)}(0)}{n!} x^n$$

For the algorithm we need the first non-zero term of the Maclaurin series of the function after the w substitution, which will be in the form $A(x)w^b$.

The final result of the limit will be the following:

$$\lim_{x\to x_0} f(x) = \lim_{x\to 0} w(x) = \begin{cases} 0 & \text{if } b > 0 \\ +\infty & \text{if } b < 0 \\ \lim_{x\to 0} A(x) & \text{if } b = 0 \end{cases}$$

2.3 Caveats and Limitations

The Gruntz algorithm, while mathematically proven to be always valid in the exp-log field of operations, has some caveats that, if not handled correctly, might prevent it from reaching the correct result, or reaching a result at all. Firstly, before computing the MRV, the limit has to be rewritten so that it tends to $+\infty$, applying some smart substitutions. Computing the correct MRV set is a complex operation, which involves recursively evaluating limits, which are proven to be of lower complexities only if simplifications are applied all throughout the process. If the MRV is of a lower comparability class compared to x, the limit has to be rewritten so that the MRV increases its comparability class, usually applying the following definition:

Definition.

$$\lim_{x\to+\infty} f(x) = +\infty \bigwedge \lim_{x\to+\infty} g(x) = +\infty \implies \lim_{x\to+\infty} f(g(x)) = +\infty$$

3 T.C.A.S.

3.1 Parsing the Expression

The first thing that TCAS does is parsing the input string. The space-separated chain of operators and operands needs to be interpreted as a concrete syntax tree [9][7], where every operator is a branch node and every symbol or value is a leaf node; this happens in multiple

step.
Firstly, the input string is tokenized, so it is split into an array of tokens based on a common delimiter which is the space character; this happens in linear time $O(n)$.
This array of tokens goes through the Polish Notation [1.2] interpreter, which in linear time $O(n)$ builds a concrete syntax tree, branching when the token is recognized to be an operator and terminating with a leaf for anything else.
This tree is incomplete though, as symbols and values are still in their string representation and have yet to be parsed; this happens in the translator, which, again in linear time $O(n)$ loops once over the concrete syntax tree and replaces every node with an *expr_tree_link*, fig. 2, fig. 5; see fig. 3 for a representation of the whole process.

3.2 Finding the MRV

Finding the MRV is a complex operation which involves multiple steps. The root node, which should always have branches, goes through *_mrv_op*, fig. 9. Dominikiz Gruntz provided a pseudo-code representation of what *_mrv_op* should do, which has been slightly altered here so that the method names correspond to their implementation:

Algorithm. *Computing the MRV set of f*

mrv (f : *exp-log function in* x)

if $x \not\triangleleft f \rightarrow RETURN(\ \{\}\ )$

elif $f = x \rightarrow RETURN(\ \{x\}\ )$

elif $f = g \times h \rightarrow RETURN(\ \{\_mrv\_max(\_mrv\_generic(g), \_mrv\_generic(h))\}\ )$

elif $f = g + h \rightarrow RETURN(\ \{\_mrv\_max(\_mrv\_generic(g), \_mrv\_generic(h))\}\ )$

elif $f = g^c \wedge c \in \mathbf{R} \rightarrow RETURN(\ \{\_mrv\_generic(g)\}\ )$

elif $f = \ln c \rightarrow RETURN(\ \{\_mrv\_generic(g)\}\ )$

elif $f = g^c \rightarrow RETURN(\ \{\_mrv\_exp(g)\}\ )$

fi

The method *_mrv_max* makes use of the definition in (2.1), with some common options coded in for performance optimization purposes; *_mrv_generic* is an internal reference to *mrv*; *_mrv_exp* is a special function that handles the MRV of exponential functions: for the generic form e^g, if $\lim_{x\to+\infty} g = \pm$ then the MRV is *_mrv_max(*$\{e^g\}$*, _mrv_generic(g))*, else $g \succ e^g$ so the MRV is *_mrv_generic*(g).

3.3 Evaluating the Result

3.4 Performance Considerations

C was chosen as the programming language for multiple reasons: it is portable, it is universal, it is fast and it is easy to use. The most commonly used implementation of the Gruntz algorithm is in SymPy [5], a Python library for symbolic mathematics.

There are almost no similarities between C and Python [6], with the former being a statically-typed [13] compiled language and the latter being a dynamically-typed [13] interpreted one. Multiple tests were run on both implementations, with the results showing an interesting pattern: fig. 4. After "warming up" SymPy, so that it loads all the dynamically-linked libraries and extensions, which was done by making it evaluate the simple limit $\lim_{x\to 0} \frac{\sin(x)}{x}$, the running time seems to scale linearly between the two systems in the first three tests. The difference is most probably due to the fact that SymPy runs in an interpreted language, so every instruction has to be parsed, evaluated and executed, while TCAS is compiled to machine code is directly and just executed.

Test-4 shows an advantage that SymPy has over TCAS: its ability to cache frequent operations; this feature, which is mostly trivial to implement, isn't ready yet, so on longer, repeated operations SymPy is able to reuse the old result and skip all the processing. This feature has some side-effects that must be considered: an increased memory usage, a small but perceivable performance reduction if the result has never been cached before, an increased complexity in memory management; if ever implemented, it should remain an opt-out feature for the cases when it might not be beneficial, *e.g.* on microcontrollers for PID [10] feedback mechanisms. Additional considerations could be made by *ptracing* [12] both executables.

References

[1] Free Software Foundation, Inc. *GNU MP. The GNU Multiple Precision Arithmetic Library*. [Online]. 2020. URL: https://gmplib.org/gmp-man-6.2.1.pdf.

[2] Free Software Foundation, Inc. *GNU MPFR. The Multiple Precision Floating-Point Reliable Library*. [Online]. 2020. URL: https://www.mpfr.org/mpfr-current/mpfr.pdf.

[3] Dominik Wolfgang Gruntz. *On Computing Limits in a Symbolic Manipulation System*. 1996. URL: https://doi.org/10.3929/ethz-a-001631582.

[4] Brian Kernighan and Dennis Ritchie. *The C Programming Language*. 1978. ISBN: 9780131101630.

[5] Aaron Meurer et al. "SymPy: symbolic computing in Python". In: *PeerJ Computer Science* 3 (Jan. 2017), e103. ISSN: 2376-5992. DOI: 10.7717/peerj-cs.103. URL: https://doi.org/10.7717/peerj-cs.103.

[6] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

[7] Wikipedia contributors. *Abstract syntax tree — Wikipedia, The Free Encyclopedia*. [Online; accessed on 26-May-2021]. 2021. URL: https://en.wikipedia.org/wiki/Abstract_syntax_tree.

[8] Wikipedia contributors. *L'Hôpital's rule — Wikipedia, The Free Encyclopedia*. [Online; accessed on 25-May-2021]. 2021. URL: https://en.wikipedia.org/wiki/L%5C%27H%5C%C3%5C%B4pital%5C%27s_rule.

[9] Wikipedia contributors. *Parse tree — Wikipedia, The Free Encyclopedia*. [Online; accessed on 26-May-2021]. 2021. URL: https://en.wikipedia.org/wiki/Parse_tree.

[10] Wikipedia contributors. *PID controller — Wikipedia, The Free Encyclopedia*. [Online; accessed on 27-May-2021]. 2021. URL: https://en.wikipedia.org/wiki/PID_controller.

[11] Wikipedia contributors. *Polish notation — Wikipedia, The Free Encyclopedia*. [Online; accessed on 25-May-2021]. 2021. URL: https://en.wikipedia.org/wiki/Polish_notation.

[12] Wikipedia contributors. *Ptrace — Wikipedia, The Free Encyclopedia*. [Online; accessed on 27-May-2021]. 2021. URL: https://en.wikipedia.org/wiki/Ptrace.

[13] Wikipedia contributors. *Type system — Wikipedia, The Free Encyclopedia*. [Online; accessed on 27-May-2021]. 2021. URL: https://en.wikipedia.org/wiki/Type_system.

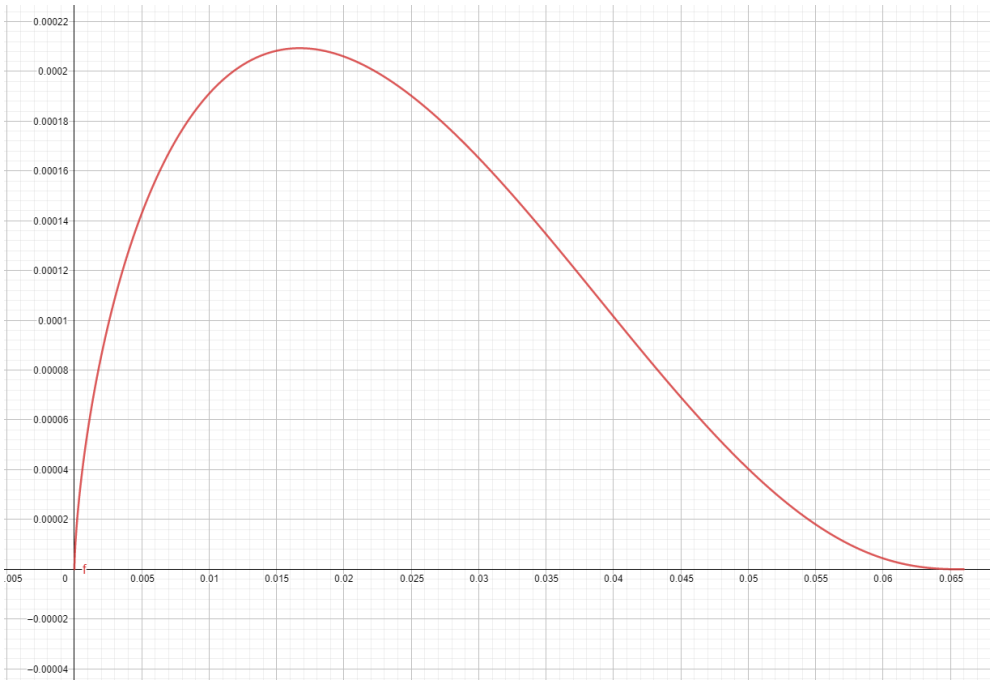

Figure 1: The graph of (1)

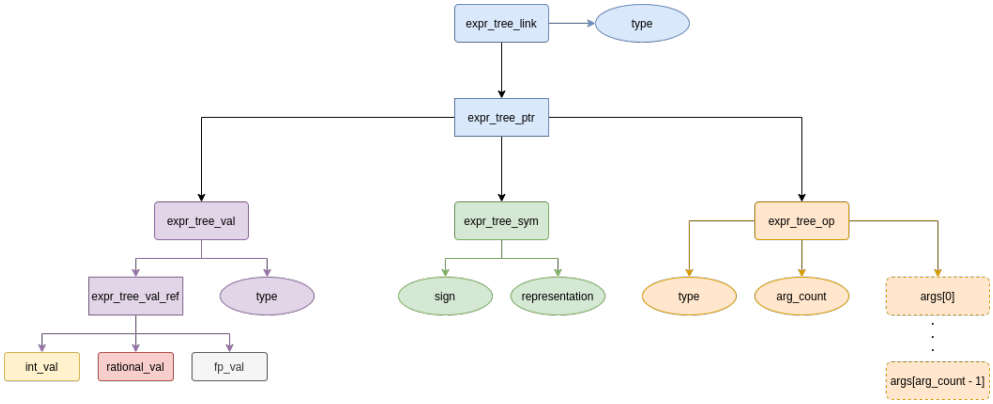

Figure 2: The entity representation of a generic node, with color-coded combinations

Figure 3: A syntactic flowchart of how the parser works

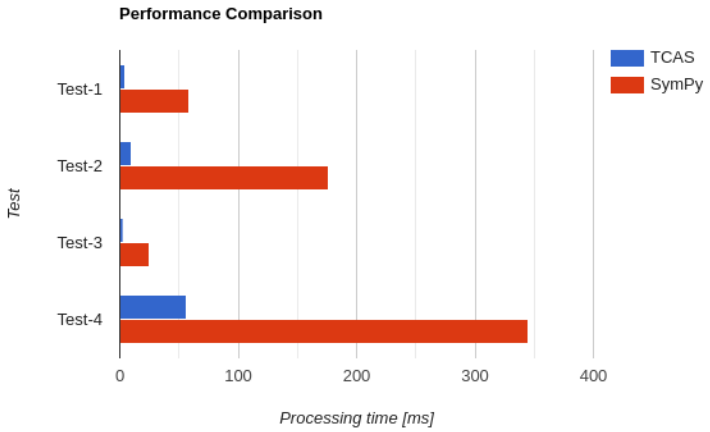

Figure 4: The graph is a performance comparison between TCAS and SymPy in four different tests.
SymPy was "warmed up" prior to the test.
Test-1 evaluated 200 times the same limit.
Test-2 evaluated the same 3 limits for 100 times.
Test-3 evaluated a very complex nested limit.
Test-4 re-ran Test-1 for 10 times

```c
#ifndef EXPR_STRUCTS_H
#define EXPR_STRUCTS_H

#include <stdio.h>
#include <stddef.h>
#include <gmp.h>
#include <mpfr.h>

enum VAL_TYPE {
  INT = 0b001,
  RATIONAL = 0b010,
  FLOAT = 0b100,
};

struct expr_tree_head {
  struct expr_tree_link *head;
};

struct expr_tree_op {
  enum OPERATOR_TYPE type;
  size_t arg_count;
  struct expr_tree_link **args;
};

struct expr_tree_val {
  enum VAL_TYPE type;
  union expr_tree_val_ref *val;
};

struct expr_tree_sym {
  char sign;
  char representation;
};

struct expr_tree_link {
  enum LINKED_TYPE type;
  union expr_tree_ptr *ptr;
};

union expr_tree_ptr {
  struct expr_tree_val *val;
  struct expr_tree_op *op;
  struct expr_tree_sym *sym;
};

union expr_tree_val_ref {
  mpz_t int_val;
  mpq_t rational_val;
  mpfr_t fp_val;
};


#endif


```

Figure 5: The code that defines the concrete syntax tree

```
struct ps_leadterm {
  int term_num;
  struct expr_tree_link *coeff;
  struct expr_tree_val *exp;
};

```

Figure 6: How the first term of the power series is represented

```
struct expr_tree_link *compute_gruntz_result(struct expr_tree_link *link) {
  struct ps_leadterm *term = _recursive_compute_lt(link);

  while (term->exp->type != INT) { //Something is wrong here, next term
    term = _recursive_lt_next(term, link);
  }

  if (mpz_cmp_si(term->exp->val->int_val, 0) > 0) { //Exp > 0
    return parse_expr("+infinity", NULL);
  } else if (mpz_cmp_si(term->exp->val->int_val, 0) == 0) { //Exp == 0
    return gruntz_eval(term->coeff);
  } else { //Exp < 0
    return parse_expr("0", NULL);
  }
}

```

Figure 7: How the result is evaluated

```
struct gruntz_mrv {
  size_t count;
  struct gruntz_expr **expr;
};

struct gruntz_expr {
  struct expr_tree_link *expr;
};

```

Figure 8: How the MRV structs are defined

```
struct gruntz_mrv *_mrv_op(struct expr_tree_link *link) {
  switch (link->ptr->op->type) {
    case PLUS:
    case MINUS:
    case TIMES:
    case DIVIDE:
      return _mrv_max(_mrv_generic(link->ptr->op->args[0]), _mrv_generic(
link->ptr->op->args[1]));
      break;
    case SQRT:
    case SIN:
    case COS:
    case TAN:
    case LN:
    case LOG10:
    case LOG2:
      return _mrv_generic(link->ptr->op->args[0]);
      break;
    case ROOT:
    case POWER:
      assert(link->ptr->op->args[1]->type == VALUE);
      return _mrv_generic(link->ptr->op->args[0]);
      break;
    case EXP:
      return _mrv_exp(link);
      break;
    default:
      assert(0);
  }
}

```

Figure 9: *_mrv_op*, refer to fig. 5 and fig. 8 for how the data types are implemented