ragni-cas - A Pure *Ruby* Automatic Differentiation Library for Fast Prototyping of Interfaces

Matteo Ragni[a]

[a]*Department of Industrial Engineering, University of Trento, 9, Sommarive, Povo di Trento, Italy*

Abstract

Ca. 100 words

Keywords: CAS, code-generation, Ruby

1. Motivation and significance

Ruby is a purely object-oriented scripting language that allows to express several programming paradigms. It was designed in the mid-1990s by Yukihiro Matsumoto (also known as *Matz*), and it is internationally standardized since 2012 as ISO/IEC 30170.

With the advent of the *Internet of Things*, a written from scratch version of the *Ruby* interpreter called *mRuby* (*eMbedded Ruby*) has been published on *GitHub* by Matsumoto in 2014. The new interpreter is a lightweight implementation aimed at both low power devices and personal computer that complies with the standard. *mRuby* has a completely new API, and it is designed to be embedded in a complex project as a front-end interface — e.g. a numerical optimization suite may use *mRuby* to get problem input definitions.

The *Ruby* code-base exposes a a large set of utilities in core and standard library. This set of tool can be furthermore expanded through libraries, also

Email address: `matteo.ragni@unitn.it` (Matteo Ragni)

known as *gems*. Even the high number of gems deployed and available, there is no library that implements a **symbolic automatic differentiation** (AD) engine that also handles some basic computer algebra routines that is cross compatible with all the different *Ruby* interpreter.

Ruby has matured its fame as a web oriented language with *Rails*, and can efficiently generate code in other languages. An AD-capable gem is the foundamental step to develop rapidly a specific code generator for well known software — e.g. IPOPT.

The library that is described in this work, is a gem implemented in pure *Ruby* code and thus compatible with all interpreter that complies with the standard, that is able to perform symbolic AD and some simple computer algebra operations. In particular the library aims at:

- be an instrument for rapid development of prototype interface for numerical algorithms — including the *mRuby* engine or exporting generated code — that can be in different languages;
- rapidly generate descriptions of mathematical models, with easy to implement workaround for numerical issues, changing on request how the code is exported, and how functions are formulated in the target language;
- creating a complete open-source CAS system for the *Ruby* language, that is be compatible with all the interpreters that comply with the standard, as a long-term ambitious impact.

This is not the first gem that tries to implement a CAS. The available computer algebra library for *Ruby* are:

Rucas gem at early stage and with discontinued developing status; it implements basic simplification routines. There is no AD method, but it is one of the milestones. The development is discontinued since 2010.

Symengine is a wrapper for the C++ library *symengine*. The back-end library is very complete, but it is compatible only with the mainstream *Ruby* interpreter. At the moment, the *SciRuby* project reports the gem as broken, and removed it from its codebase. From a direct test, when performing AD of a function, the engine returns always `nil`.

2. Software description

2.1. Software Architecture

ragni-cas is an object oriented AD gem that supports some simple computer algebra routines such as *simplifications* and *substitutions*. When gem is required, automatically overloads some methods of the `Fixnum` and `Float` classes, to make them compatible with the foundamental symbolic objects.

Each symbolic function is an object modeled by a class, that inherits from a common virtual ancestor: `CAS::Op`(operation). An operation encapsulates sub-operations recursively, building a linked graph, that is the mathematical equivalent of function composition:

$$(f \circ g) \tag{1}$$

When a new operation is created, it is appended to the graph. The number of branches are determined by the parent container class of the current symbolic function. There are three possible containers. Single argument functions — e.g. $\sin(\cdot)$ — have as closest parent the `CAS::Op` class, that links to one sub-graph. Functions with two arguments — e.g. difference or exponential function — inherit from `CAS::BinaryOp`, that links to two subgraphs. Functions with arbitrary number of arguments — e.g. sum and

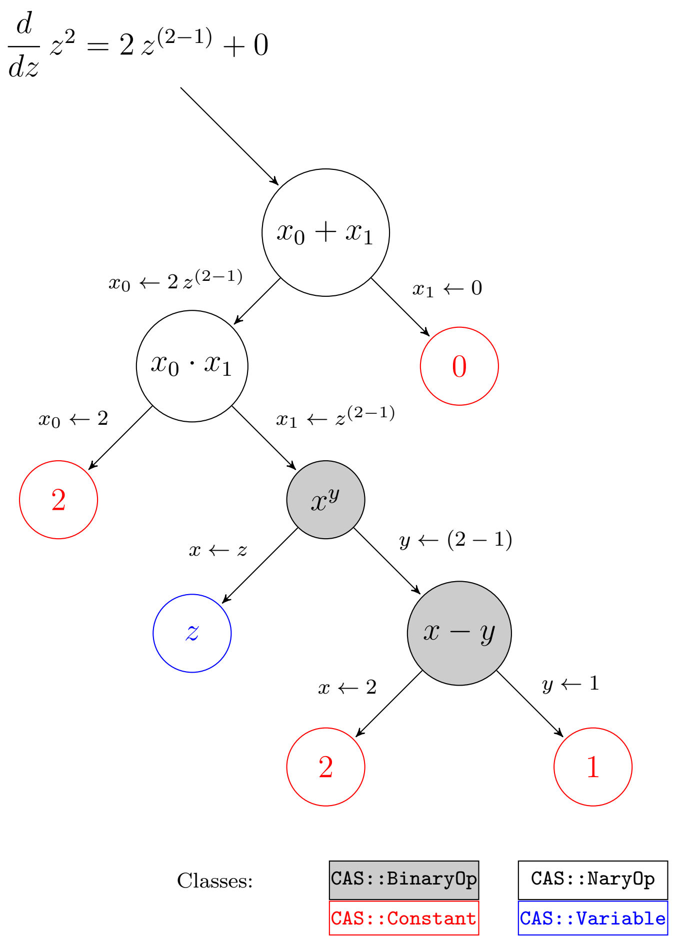

Figure 1: Example graph from the first function reported in script **??**

product — have as parent the `CAS::NaryOp`[1], that links to an arbitrary number of subgraph. Figure 2.1 contains an example of graph. The different kind of containers allows to introduce some properties like *associativity* and *commutativity*. Each container exposes the subgraphs as instance properties. Containers structure is shown in Figure 2.1.

Terminal leaf of the graph are the two classes `CAS::Constant` and `CAS::Variable`. The first is a node for a simple numerical value, while the latter

[1]Please note that this container is still at experimental stage

represents an independent variable, that can be used to perform derivatives and evaluations. As for now, those nodes are only scalar numbers, with plans to define also the vectorial and matricial extensions in the next major release.

Automatic differentiation (`CAS::Op#diff`) crosses the graph until it reaches the ending node. The terminal node is the starting point for derivatives accumulation, the mathematical equivalent of the chain rule:

$$(f \circ g)' = (f' \circ g)\ g' \tag{2}$$

The recursiveness is used also for simplifications (`CAS::Op#simplify`), substitutions (`CAS::Op#subs`) and evaluations (`CAS::Op#call`).

2.2. Software Functionalities

The main functionality of the library is the **AD**, that can be performed with respect to an independent variable (`CAS::Variable`), even for implicit functions. The differentiation is done by a method of the `CAS::Op`, having a `CAS::Variable`as argument:

```
\label{code:example-diff}
x = CAS.vars 'x'            # creates a variable
f = x ** 2 + 1              # define a symbolic expression
f.diff(x)                   # derivative w.r.t. x
# => 2 * x ^ (2 - 1) + 0
g = CAS.declare :g, f       # creates implicit function
g.diff(x)                   # derivative w.r.t. x
# => (2 * x ^ (2 - 1) + 0) * Dg[0](x ^ 2)
```

Resulting graph still contains a zero, since **simplifications** are not executed automatically. Each node of the graph contains some rules for simplify itself. Simplification are called recursively inside the graph, exactly like AD, bringing the strong limitation of not handling simplifications that come from *heuristic expansion* of sub-graphs — e.g. simplification through the use

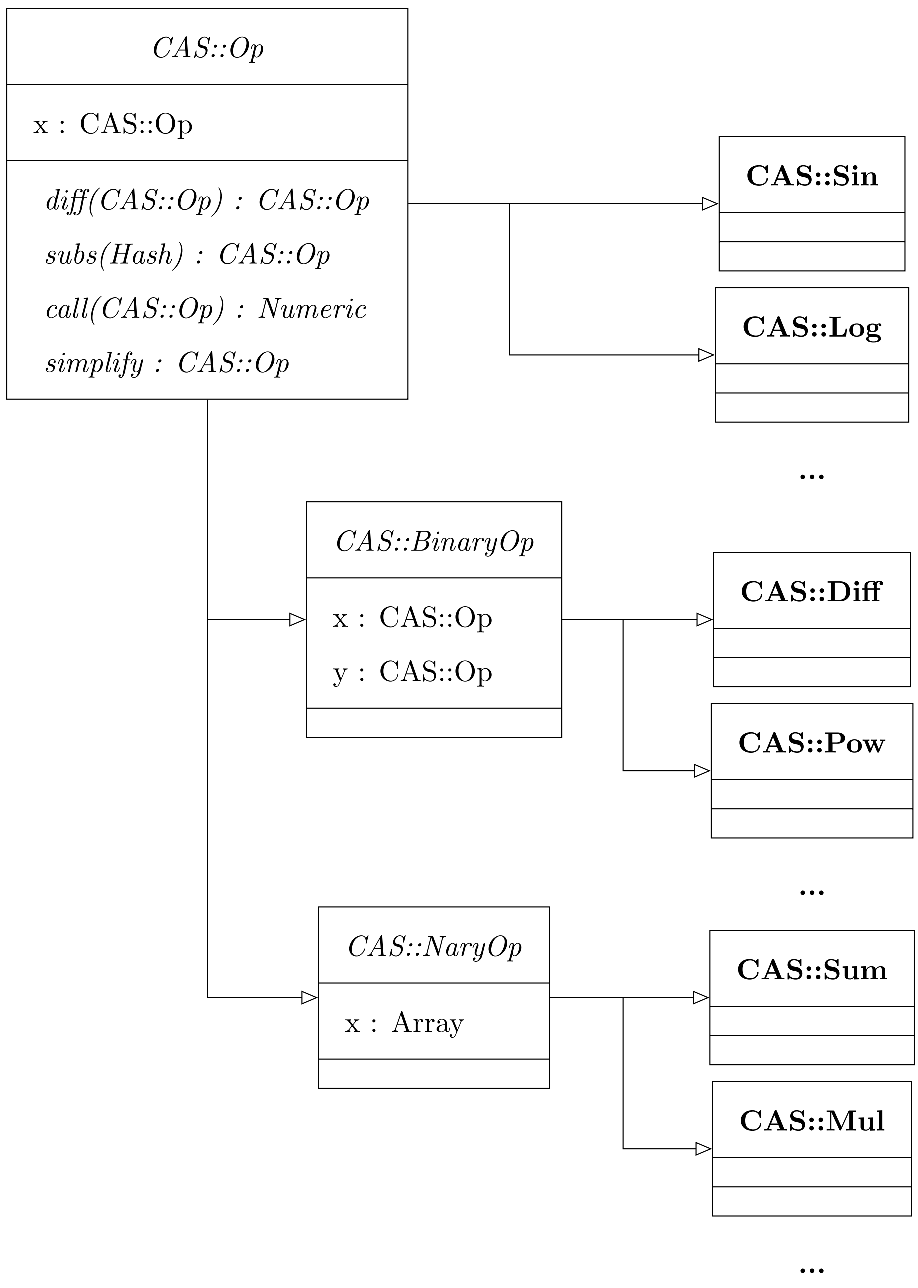

Figure 2: Simplified version of classes interface and inheritance

of trigonometric identities. Those simplification can be achieved manually using **substitutions**.

```
\label{code:example-subs}
x, y = CAS::vars 'x', 'y'          # creates two variables
f = CAS.log( CAS.sin( y ) )        # symbolic expression
f.subs  y: CAS.asin(CAS.exp(x))    # perform substitution
f.simplify                         # simplify expression
# => x
```

The graph can be **evaluated** substituting defining some values for the independent variable in a feed dictionary. The graph is recursively reduced to a single numeric value:

```
\label{code:example-call}
x = CAS::vars 'x'            # creates a variable
f = x ** 2 + 1               # define a symbolic expression
f.call x: 2                  # evaluate for x = 2
# => 5
```

Symbolic functions can be used to create expressions — e.g. $f(\cdot) \geq g(\cdot)$ — or piecewise functions — e.g. $\max(f(\cdot), g(\cdot))$:

```
x = CAS::vars 'x'
f = x ** 2
g = 2 * x + 1
f.greater_equal g
# => ((x)^(3) >= ((2 * x) + 1))
CAS::max f, g
# => (((x)^(3) >= ((2 * x) + 1)) ? (x)^(3) : ((2 * x) + 1))
```

Expression are stored in a special container class, called `CAS::Condition`.

The library is developed explicitly for **generation of code**, and in some case also **metaprogramming**. Expressions, once manipulated, can be easily exported as source code (in a defined language —i.e. the following example in standard *Ruby* code):

```
x = CAS::vars 'x'              # creates a variable
```

```ruby
f = CAS::log(CAS::sin(x))     # define a symbolic function
# => Math::log(Math::sin(x))
```

the same function can be also used to create directly a callable *closure*, as a `Proc` object:

```ruby
proc = f.as_proc              # exports callable lambda
proc.call 'x' => Math::PI/2
# => 0.0
```

Must be noted that parsing the graph creates a snapshot, and any further modification to the expression will not update the callable object. This drawback is balanced by faster execution time of the `Proc`: when a graph needs only to be evaluated in a iterative algorithm, and not to be manipulated, transforming it in a *closure* reduces the execution time per loop.

3. Illustrative Examples

4. Impact

5. Conclusions

Acknowledgements

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

Current code version

Nr.	Code metadata description	Please fill in this column
C1	Current code version	0.0.0
C2	Permanent link to code/repository used for this code version	github.com/MatteoRagni/cas-rb & rubygems.org/gems/ragni-cas
C3	Legal Code License	MIT
C4	Code versioning system used	*git* (GitHub)
C5	Software code languages, tools, and services used	*Ruby*
C6	Compilation requirements, operating environments	*Ruby* $\geq$ 2.*x*, *pry* for testing console (optional)
C7	If available Link to developer documentation/manual	rubydoc.info/gems/ragni-cas
C8	Support email for questions	info@ragni.me

Table 1: Code metadata (mandatory)