Writing Converters for the Function Convert Package

Barton Willis

February 23, 2026

Contents

1 Introduction and prerequisites

Maxima’s `function convert` package [1] does semantic function-to-function conversions on Maxima [2] expressions. This document is a tutorial on how to extend the mathematical capabilities of this package by writing your own converters. To do this, you will need

- to know how to use Maxima and how to use the function convert package,
- to know how to program Maxima using Common Lisp.

Source Note Some text in this tutorial is taken from, or adapted from, the project’s `README.md` on GitHub [1], with the permission of the author.

2 Features

We give an overview of the features of the `function convert` package. We start with basic usage; here an example of a converter that replaces each subexpression of a Maxima expression of the form $\mathrm{erfc}(X)$ by $1 - \mathrm{erf}(X)$, where erf and erfc are error functions and X matches any expression. This converter is built-in to the `function convert` package. To use it, input:

```
(%i1) function_convert(erfc = erf, erfc(x));
(%o1)                              1 - erf(x)
```

Of course, the input can be any expression, including nested calls to erf; for example

```
(%i2) function_convert(erfc = erf, x*erfc(x + erfc(z)));
(%o2)                      x (erf(erf(z) - x - 1) + 1)
```

The function `function_convert` walks an expression tree and replaces function calls according to well-defined rules. In that sense, it is straightforward code of a kind repeated many times in Maxima. It is not a pattern matcher or a general rewrite engine.

The package is implemented in Common Lisp. It has been tested using Clozure Common Lisp [3] version 1.13 and SBCL [4] version 2.4.7.

2.1 Naming and the alias system

In these examples, the left-hand side of the conversion rule `erfc = erf`, is the *source function* and the right-hand side is called `erf` the *target function*. The rule `erfc = erf` does not literally replace the source function with the target function, but it instead applies the identity $\mathrm{erfc}(x) = 1 - \mathrm{erf}(x)$ to an expression, replacing the left side with the right side.

The name of the target function is best thought of a mnemonic for recalling the identity $\mathrm{erfc}(x) = 1 - \mathrm{erf}(x)$, not as a literal function. Actually, the name of the target function could be any Maxima symbol; for example, the rule could be named `erfc = erfc_to_erf`.

Similarly, the alias system also allows the source function to be a mnemonic as well. This feature is useful for naming a converter for an identity such as

$$\Gamma(z)\,\Gamma(1-z) = \pi/\sin(\pi z)\,. \tag{1}$$

Here the left-hand-side of this identity is a product, so the source function for this identity must be multiplication (internally `mtimes`), not the gamma function. But to a user the conversion rule `"*" = sin` may seem unnatural. The alias system allows the converter for this identity to have a much more natural name. The converter for identity Eq. (1) is built-in to the `function convert` package; the alias for this converter is `gamma = sin`.

A macro takes care of the messy details of defining an alias, along with other housekeeping duties.

2.2 Self-documentation feature

Every converter function should have a doc string. The doc string is available to the user via the function `describe_converter`; for example

```
    (%i1) describe_converter(sinc = sin);
Converter: %SINC = %SIN
Function: #<FUNCTION FUNCTION-CONVERTER-%SINC-%SIN>
Type: built-in

Docstring:
  Convert sinc(x) into sin(x)/x.
```

If a user-package properly includes a doc string for a converter, that documentation will be available via the function `describe_converter` once the package is loaded. The name of the converter, in this case `FUNCTION-CONVERTER-%SINC-%SIN`, is automatically generated by a macro that is used to define a converter–the function name of a converted is unlikely to be useful to a user.

2.3 Class Keys and Class-Based Dispatch

Converter functions often need to apply the same transformation to an entire family of related operators. For example, a converter that rewrites trigonometric functions into exponential functions should apply uniformly to all six trigonometric functions. Writing six separate converters is tedious and error-prone. The *class key* mechanism provides a solution.

Purpose of Class Keys A *class key* is a symbolic tag (such as `:trig` or `:hyperbolic`) that represents a category of operators. The converter system uses these tags to perform *class-based dispatch*: a single converter registered for a class key is automatically applied to every operator belonging to that class.

Class keys classify operators, not expression types. For example, the class `:algebraic` contains the operators `mplus`, `mtimes`, and `mexpt`. But if the head operator of an expression belongs to the class `:algebraic`, that does not imply that the expression itself is algebraic.

The Class Table The mapping from class keys to operators resides in the global `*converter-class-table*`. This table is

```
(defparameter *converter-class-table*
  '((:trig        . (%sin %cos %tan %sec %csc %cot))
    (:hyperbolic  . (%sinh %cosh %tanh %sech %csch %coth))
    (:inv_trig    . (%asin %acos %atan %asec %acsc %acot))
    (:inv_hyperbolic . (%asinh %acosh %atanh %asech %acsch %acoth))
    (:exp         . (mexpt))
    (:gamma_like  . (%gamma %beta %binomial %double_factorial
                     mfactorial $pochhammer))
    (:bessel  . (%bessel_j %bessel_y %bessel_i %bessel_k
                 %hankel_1 %hankel_2 %airy_ai %airy_bi
                 %airy_dai %airy_dbi))
    (:algebraic . (mplus mtimes mexpt))
    (:inequation . (mequal mlessp mleqp mnotequal mgreaterp mgeqp
```

```
                  $notequal $equal))
    (:logarithmic . (%log)))
  "Mapping from class keys to lists of operator symbols.")
```

Each entry associates a class key with the list of operators that belong to that class. The system assumes that *An operator belongs to exactly one class*. This ensures unambiguous dispatch: when the converter system encounters an operator, it can determine its class uniquely and apply the appropriate class-level converter.

How Class Dispatch Works When the converter system processes an expression, it identifies the operator at the head of the form. It then:

(a) looks up the operator in `*converter-class-table*`;
(b) determines the corresponding class key (e.g., `:trig`);
(c) checks whether a converter is registered for that class;
(d) if so, applies that converter to the entire expression.

Why Class Keys Matter Class keys provide:

- **Abstraction**: one converter per conceptual family.
- **Extensibility**: adding a new operator to a class requires no changes to existing converters.
- **Uniformity**: guarantees consistent behavior across related operators.
- **Clarity**: the class table documents the structure of the operator space.

2.4 Chaining multiple conversions

To apply two or more conversions, put the converters into a list; for example

```
(%i1) function_convert(['sinc = 'sin, 'sin = 'exp], sinc(x)^2);
(%o1) -((%e^(%i*x)-%e^-(%i*x))^2/(4*x^2))
```

Changing multiple conversions is handled by the function `function_convert`; a writer of a converter doesn't need to do anything to allow chaining to work. Converters are applied left-to-right

2.5 Implicit Dispatch via Breadth-First Search

The converter system automatically discovers multistep conversion chains. If a converter is defined from f to g and another from g to h, the system is able to convert f to h without any additional user-defined rule. In other words, there is no need to define a direct converter f to h.

This behavior is enabled by a breadth-first search (BFS) over the converter graph. The search locates the shortest available path between two operators and composes the corresponding converters. Writers of converters only need to know that `function convert` uses BFS, not anything about the internals of this mechanism.

2.6 Errors and Debugging

When a converter is redefined, Maxima will issue a warning, and when a converter doesn't exist, Maxima returns the input unchanged, but doesn't issue a warning; for example

```
(%i2) function_convert(f = g, a+b);
(%o2)                                b + a
```

When writing a new converter, if a converter doesn't trigger, the problem is likely due to either a typo or some confusion between Maxima nouns and verbs. If a converter that doesn't trigger correctly, try tracing the Common Lisp function `lookup-converter`. From there, you might be able to diagnose the problem.

3 Writing converter functions

A macro `define-function-converter` takes care of many of the housekeeping details in writing a converter, naming the internal function that does the conversing, including optionally marking the converter as built-in, and gathering the doc string for the converter to use a user-documentation. The macro also automatically handles the alias system.

The function `function_convert` does the work of mapping the converter over the expression tree; this allows the converter code to be relatively simple. We'll begin by examining the source code of the `erfc = erf` converter.

3.1 The erfc to erf converter

The source code for the erfc to erf converter is

```
(define-function-converter (%erfc %erf) (op x)
    :builtin
  "Convert erfc(x) into 1 - erf(x)."
  (declare (ignore op))
  (let ((z (car x)))
    (sub 1 (ftake '%erf z))))
```

The arguments `%erf %erfc` are the source and target functions, respectively. For this converter, the argument `op` is bound to the source function `%erfc`, but this converter doesn't use this argument, so the converter code should declare to ignore it. The argument `x` is a Common Lisp list of the argument to the function `%erfc`.

The optional keyword argument `:builtin` declares that this converter cannot be deleted using user-level functions. The doc string is optional, but important because supplies the converter with documentation. The optional keyword must be placed before the doc string.

The mathematical part of the code, that is

```
  (let ((z (car x)))
    (sub 1 (ftake '%erf z)))
```

implements the identity $\mathrm{erfc}(x) = 1 - \mathrm{erf}(x)$. Notice that since `x` is a Common Lisp list of the argument to `%erfc`, it is necessary to apply `%erf` to the car of `x`, and not to `x` itself. Alternatively, if you prefer, this bit of code can be changed to

```
      (sub 1 (fapply '%erf x))
```

All the other details are handled by the macro `define-function-converter`.

3.2 Alias Dispatch

As we discussed before, both the source and the target functions of a converter can be given an alias. This can be useful when the actual source function is a poor mnemonic for the converter name.

Again, the macro `define-function-converter` takes care of the housekeeping for defining an alias. The best way to explain how to use alias dispatch is by an example. Here we give a converter that expands explicit powers, but not products of sums. The source function must be `mexpt`, but we'll make `power = expand` an alias for `"^" = expand`. This is how it is done:

```
(define-function-converter ((mexpt $expand) ($power $expand)) (op x)
  ($expand (fapply 'mexpt x)))
```

There are two ways to invoke this converter:

```
(%i1) function_convert("^" = expand, (a+b)*x + (1+x)^2);
(%o1) x^2+(b+a)*x+2*x+1

(%i2) function_convert(power = expand_powers, (a+b)*x + (1+x)^2);
(%o2) x^2+(b+a)*x+2*x+1
```

3.3 Class Key Converter Dispatch

Class-key converters allow a single definition to handle an entire family of related operators. The following example illustrates this idea using the trigonometric class `:trig`. Any operator listed under the `:trig` class key in `*converter-class-table*` will be handled by the same converter, without the need to define six separate rules. For this converter, we finally have a reason for the `op` argument–this argument will be bound to anyone of the six trigonometric functions. The main code uses a `case` statement to handle each of the six cases.

```
(define-function-converter (:trig $sin_cos) (op x)
  :builtin
  ;; Convert all six trigonometric functions to sin/cos form.
  (let ((z (first x)))
    (case op
      (%sin (ftake '%sin z))
      (%cos (ftake '%cos z))
      (%tan (div (ftake '%sin z) (ftake '%cos z)))
```

```
    (%sec (div 1 (ftake '%cos z)))
    (%csc (div 1 (ftake '%sin z)))
    (%cot (div (ftake '%cos z) (ftake '%sin z)))
    (t (ftake op z)))))
```

This converter is registered for the class key `:trig`, not for any individual operator. When the dispatch system encounters an expression such as `(%tan x)` or `(%csc x)`, it determines that the operator belongs to the `:trig` class and invokes the converter above.

The body of the converter receives two arguments: the operator `op` and its argument list `x`. The converter then rewrites each trigonometric function into its sine–cosine form. Because the class key handles the grouping, the converter itself contains only the mathematical logic; it does not need to know how many operators belong to the class or where they appear in the system.

Class-key converters therefore provide a uniform and extensible mechanism for handling families of related functions. Adding a new trigonometric operator to the system requires only updating the class table; the converter itself does not need to be modified.

3.4 A more advanced converter

In this section, we show to to build a converter that rewrites explicit products of the form $x\text{signum}(x)$ as $|x|$. The converter does not rewrite $\text{signum}(x)$ itself, because the pattern $x\text{signum}(x)$ is not a subexpression of $\text{signum}(x)$; only explicit products matching that pattern are transformed–this is what makes writing the converter somewhat tricky. The converter logic needs to scan the arguments of each product and gather the terms that have signum as their head operator. For each of these terms, the converter does a rational substitution (`ratsimp`) of the form `ratsubst(abs(s), s*signum(s), e)` on the expression `e`, that must be a product. The definition is:

```
(define-function-converter ((mtimes mabs) (%signum mabs)) (op x)
  :builtin
  "Convert subexpressions of the form X*signum(X) into abs(X).  This converter
does not rewrite signum(X) itself, since X*signum(X) is not a subexpression
of signum(X); only explicit products matching that pattern are transformed."
  (let* ((e  (fapply op x))
         (ll (xgather-args-of e '%signum)))
    (dolist (lx ll)
      (let ((s (car lx)))
        (setq e ($ratsubst (ftake 'mabs s)
                           (mul s (ftake '%signum s))
                           e))))
    ($expand e 0 0)))
```

How the converter works

(a) The converter receives the head operator `op` and its argument list `x`. The call `(fapply op x)` reconstructs the full expression e.

(b) The function `xgather-args-of` scans e for occurrences of `%signum` and returns a list of matches. Each element of this list contains the argument X of a subexpression signum(X).
(c) For each such argument s, the converter constructs the product $s \cdot \text{signum}(s)$ and replaces it with abs(s) using `$ratsubst`. Only explicit products are rewritten; no algebraic inference is performed.
(d) After all substitutions, the expression is expanded with `$expand` to restore a canonical algebraic form.

This converter is intentionally conservative: it rewrites only the literal pattern $X \cdot \text{signum}(X)$, never the function signum itself, and never expressions that are merely algebraically equivalent.

4 Miscellaneous

We end with a few comments about the `function convert` package that are not directly related to the process of writing conveter functions.

4.1 Motivation

Many systems (Maple [5], Mathematica [6], SymPy [7]) provide built-in expansions or rewrite mechanisms, but Maxima uses an alphabet soup of functions that perform semantic function-to-function conversions; examples include `'makefact'` and `'makegamma'`. In other cases, transformations are controlled by option variables—for example, `'expintrep'`.

These names are easy to forget and are not always easy to locate in the user documentation. The `function convert` package may offer a simple, uniform, and user-extensible way to perform such conversions.

4.2 Using `'defrule'` as an alternative to `function convert`

Maxima's pattern-base `'defrule'` tool is an alternative to using `function convert` . A simple example

```
matchdeclare(aa,true)$
defrule(sinc_rule, sinc(aa), sin(aa)/aa);
sinc_rule:sinc(aa)\-\>sin(aa)/aa
apply1(sinc(sinc(x)), sinc_rule);
(x*sin(sin(x)/x))/sin(x)
```

This method works well, especially for single use function-to-function conversions. But a `'kill(all)'` removes all rules defined by `'defrule'` and it still relies on an alphabet soup of functions. Changing Maxima to allow rules that cannot be removed by `'kill(all)'` is, of course, a small matter of programming [8], but it would violate a long-standing design principle that users must always be able to return the system to a clean, rule-free state.

Additionally, it is difficult to define rules that need to match two or more terms in a sum or product. The `function convert` package has several converters that do just that. Often these converters internally rely on the `ratsubst` to do semantic substitutions–something that is difficult for a rule defined by `'defrule'`. Finally, the package has at least one built-in rule that is difficult to fully duplicate using `'defrule'`:

```
(%i1) function_convert('gamma = 'sin, 20252*x*gamma(x)*gamma(1-x) < %pi);
(%o1) (20252*%pi*x)/sin(%pi*x) < %pi
```

References

[1] B. Willis, "function_convert: Semantic function-to-function conversions for maxima," https://github.com/barton-willis/function_convert, 2026, gitHub repository.

[2] Maxima Development Team, *Maxima, a Computer Algebra System*, Maxima, 2026, version 5.47.0 or later. [Online]. Available: https://maxima.sourceforge.io/

[3] Clozure Associates, *Clozure Common Lisp*, Clozure, 2026, version 1.13 or later. [Online]. Available: https://ccl.clozure.com/

[4] SBCL Development Team, *SBCL: Steel Bank Common Lisp*, Steel Bank Common Lisp, 2026, version 2.x. [Online]. Available: https://www.sbcl.org/

[5] Maplesoft, "The convert function," https://www.maplesoft.com/support/help/maple/view.aspx?path=convert, accessed 2026-02-23.

[6] W. Research, "Rules," https://reference.wolfram.com/language/guide/Rules.html, accessed 2026-02-23.

[7] S. D. Team, "Term rewriting," https://docs.sympy.org/latest/modules/rewriting.html, accessed 2026-02-23.

[8] B. A. Nardi, *A Small Matter of Programming: Perspectives on End User Computing*. Cambridge, MA: MIT Press, 1993.