Computer-Automated Rewriting

Brett Saiki
University of Washington
ENGR 231

October 2021

While mainly relevant to computer science, mathematics, and logic, the technique of "rewriting" is universal across many different fields. Informally, rewriting is just the act of replacing an object A with another object B. A couple of naive examples that roughly illustrate this concept include replacing a word in a sentence with another word with a different connotation, or using $(x+y)^2$ instead of $(x+y)(x+y)$ since the expression is more compact.

Name	Rule
associativity	$(a+b)+c \to a+(b+c)$
commutativity	$a+b \to b+a$
additive identity	$a+0 \to a$
additive inverse	$a+(-a) \to 0$

Figure 1: Common mathematical rewrite rules. Because each rule represents an equivalance, they are technically bidirectional. In abstract algebra, these rules hold true for some domain D, e.g. $\mathbb{N}$, $\mathbb{R}$, etc. if addition and D form an abelian group.

More formally, given a domain D of objects (words, symbols, numbers, etc.), *rewriting* is the act of replacing some object $a \in D$ with another object $b \in D$. Rewriting is performed using a set of *rewrite rules*, each usually denoted $a \to b$. Rewriting usually preserves some property, e.g. equality, but this need not be always true (Figure 1).

Rewriting is abundant across various fields due to its inextricable link to logic and mathematics. It is an important tool for improving the efficiency of algorithms and computation. In particular, computer-automated rewriting has current applications in real-world domains like machine learning [1], computer-aided design [2], and numerical computation [3]; and the technique will become increasingly important in the coming decades.

Background

Rewriting is most important when performing optimization under a series of constraints, especially when improving algorithms or programs. A naive but powerful example is numerically computing the exponential function. Calculus tells us that the exponential function is given by

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

With no constraints, this representation is the ideal formula for e^x since it *exactly* computes the exponential function. However, someone using a computer trying to compute something like e^2 would probably not want to wait forever for the answer. Thus, with a time constraint, we should only compute the series with a given number of terms.

$$e^x \approx \sum_{n=0}^{N} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2} + \ldots + \frac{x^N}{N!}$$

For modern implementations on computers, however, even this method is too slow and inaccurate for large values. Instead, most algorithms use helpful identities to quickly compute e^x on a narrow interval using a quickly converging polynomial with few terms (Figure 2). Similar techniques have been to applied to other math functions such as $\sin x$, $\log x$, x^y to efficiently compute their numerical results.

Step	Description
1. Range reduction	Find an integer m such that $e^x = e^{(r+m\log 2)} = 2^m e^r$. Then, $\lvert r\rvert \leq \log 2/2 \approx 0.34658$
2. Remez polynomial	$e^r \approx 1 + r + \frac{rF(r)}{2-F(r)}$ $F(r) = r - P_1r^2 - P_2r^4 - P_3r^6 - P_4r^8 - P_5r^{10}$ $P_1 = 1.66666666666666019037 \times 10^{-1}$ $P_2 = -2.77777777770155933842 \times 10^{-3}$ $P_3 = 6.61375632143793436117 \times 10^{-5}$ $P_4 = -1.65339022054652515390 \times 10^{-6}$ $P_5 = 4.13813679705723846039 \times 10^{-8}$
3. Normalization	$e^x = 2^m e^r$

Figure 2: A method of computing the exponential function with double-precision (64-bit) floating-point numbers from OpenLibm, an open source math library currently used by the Julia language [4].

These two changes to computing the exponential functions demonstrate how algorithms can be "improved" through rewriting. However, the age-old question with small examples is: is this relavant in the real world?

Extend this exercise to every task done on computers today and identify additional constraints that may be important (accuracy, power consumption, number of computers in a cluster, etc.) and the value of rewriting becomes apparent. Just imagine if every computer in the world switched to using an inefficient implementation of e^x or $\sin x$. Inefficient methods means additional time spent on a problem and wasted resouces.

Applications

In the past, finding better ways of rewriting programs was often a manual, arduous task. Experts had to think about possible inefficiencies, test their theories, publish their results, and apply the results to existing tools. Compilers, programs that convert source code into executable code, are prime examples of tools that can automate rewriting, many of which are decades old. Thus, computer-automated rewriting remained confined to few domains like compilers.

This has led to an interesting question: what other domains might computer-aided rewriting be useful in? In recent years, there has been significant research about using rewriting in domains such as machine learning (ML), computer-aided design (CAD), and numerical computation.

```
(Union
  (Rotate [0, 0, 120] (Translate [1, −0.5, 0] (Cuboid [10, 1, 1])))
  (Scale [10, 1, 1] (Translate [1, −0.5, 0] (Cuboid [1, 1, 1])))
  (Rotate [0, 0, 300] (Translate [1, −0.5, 0] (Cuboid [10, 1, 1])))
  (Scale [5 5 1] (Cylinder [1 1]))
  (Translate [−1, 0.5, 0] (Scale [−1, −1, 1] (Cuboid [10, 1, 1])))
  (Rotate [0, 0, 240] (Translate [1, −0.5, 0] (Cuboid [10, 1, 1])))
  (Rotate [0, 0, 60] (Translate [1, −0.5, 0] (Cuboid [10, 1, 1]))))
```

```
(Union
  (Cylinder [1 5])
  (Union
    (Rotate [0 0 0] (Translate [1, −0.5, 0] (Cuboid [10, 1, 1])))
    (Rotate [0 0 60] (Translate [1, −0.5, 0] (Cuboid [10, 1, 1])))
    (Rotate [0 0 120] (Translate [1, −0.5, 0] (Cuboid [10, 1, 1])))
    (Rotate [0 0 180] (Translate [1, −0.5, 0] (Cuboid [10, 1, 1])))
    (Rotate [0 0 240] (Translate [1, −0.5, 0] (Cuboid [10, 1, 1])))
    (Rotate [0 0 300] (Translate [1, −0.5, 0] (Cuboid [10, 1, 1])))))
```

```
(Union
  (Cylinder [1 5 5])
  (Fold Union
    (Tabulate (i 6)
      (Rotate [0, 0, 60i]
        (Translate [1, −0.5, 0]
          (Cuboid [10, 1, 1]))))))
```

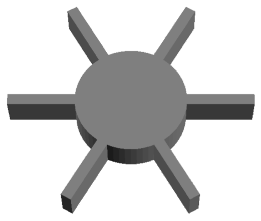

Figure 3: Three equivalent representations of a wheel CAD model [5]. The first obfuscates the model that it represents, perhaps created directly from users copying and manipulating individual components in a CAD software. The second is more ideal since it makes the structure of the model clear. The third is a compact representation that highlights the repetitive nature of the components.

Roughly speaking, machine learning takes in sample data and builds a statistical model, so it can predict future outcomes or classify data. In a sense, maching learning models are just programs that take in data and output an answer to a question, so specialized compilers have been created to optimize these models based on speed, resource consumption, and accuracy concerns. The Apache TVM compiler, a project that began at the University of Washington, is an example of such a tool.

In CAD, rewriting has proven useful in manipulating Constructive Solid Geometry (CSG) expressions. CSG expressions descibes how to construct a complex shape, e.g. a cup, from smaller primitive objects, e.g. cube, cylinder. To illustrate the utility of rewriting for CAD, imagine a wheel with some number of identical, evenly-spaced spokes. CAD software may output a CSG representation of this wheel, where the wheel is the union (combination) of many spoke objects, a hub object, and a rim object. However, this representation may obfuscate the fact that each spoke is identical but translated and rotated in space. Thus a more efficient CSG representation will notate the repetition of the spoke object and make it easier for CAD software to manipulate it (Figure 3).

$$\begin{array}{rcl} \sqrt{x+1}-\sqrt{x} & \to & \dfrac{1}{\sqrt{x+1}+\sqrt{x}} \\ \sin(x+\varepsilon)-\sin x & \to & \cos x \sin\varepsilon + \sin x(\cos\varepsilon - 1) \\ (e^x-2)+e^{-x} & \to & x^2+\dfrac{x^4}{12}+\dfrac{x^6}{360}+\dfrac{x^8}{120160} \end{array}$$

Figure 4: A few example rewrite rules for numerical computation. For each rule, the naive error-prone version is on left, and the more accurate version is on right. These rewrites assume computing with standard floating-point numbers, i.e. inexact or decimal numbers.

In numerical computation, rewriting is especially important since mathematical operations on inexact numbers, e.g. 3.14, requires rounding which can lead to inaccurate results (Figure 4). Thus every-day equations like the quadratic equation can produce widely wrong answers depending on the inputs we use. A number of tools in recent years apply automated rewriting to improve the accuracy of numerical programs that in the past would require a skilled numerical analyst to optimize by hand.

Research

My main area of research is in computer-automated rewriting, specifically in building tools for improving numerical computation. In particular, I am interested in the following issue. With rewriting being applied to different domains, the need for efficient, verified rewrite rules is increasingly necessarily, but how do you automatically find and validate rewrite rules for a particular domain?

More formally: given a grammar G containing a set of objects D and operators $f_1, f_2, \ldots$, an evaluator E that can produce the result of expressions in G, and a validator that can prove that $e_1 \to e_2$ for $e_1, e_2 \in G$ is a valid rewrite, can rewrite rules be discovered? A tool I am helping develop, called Ruler [5], produces rewrite rules for existing domains like boolean rules, e.g. a AND false $\to$ false, and rational rules, e.g. $a + b \to b + a$. However, other domains like real numbers have proven to be difficult to synthesize rewrite rules for.

There are a few possible domains that would benefit from this tool. One example is uncommon number systems like low-precision numbers used in ML and graphics (Google's bfloat16 numbers and half-precision numbers on Nvidia GPUs). These number systems may require custom rewrite rules to balance accuracy and speed since computations can behave nonsensically with few significant digits. Another interesting domain is circuit design: circuits can be redesigned to optimize accuracy, chip area, and power consumption in hardware using rewriting techniques traditionally used for software. The rearrangement of wires, gates, circut components can greatly affect the performance of a computer chip.

Conclusion

In computer science, rewriting is an important tool in program optimization and is one of the primary tools for achieving efficiency gains. However, it is rarely known about outside of research communities. Even so, work on the subject continues, improving everything from circuits to software to CAD and more. As computers become more powerful, rewriting will be increasingly applied to new domains in innovative ways and will become an increasingly important technique in the near future.

References

[1] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: end-to-end optimization stack for deep learning," *CoRR*, vol. abs/1802.04799, 2018. [Online]. Available: http://arxiv.org/abs/1802.04799

[2] C. Nandi, M. Willsey, A. Anderson, J. R. Wilcox, E. Darulova, D. Grossman, and Z. Tatlock, "Synthesizing structured cad models with equality saturation and inverse transformations," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 31–44. [Online]. Available: https://doi.org/10.1145/3385412.3386012

[3] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," *SIGPLAN Not.*, vol. 50, no. 6, p. 1–11, Jun. 2015. [Online]. Available: https://doi.org/10.1145/2813885.2737959

[4] JuliaLang, "OpenLibm," http://github.com/JuliaMath/openlibm, 2011 - 2021.

[5] C. Nandi, M. Willsey, A. Zhu, Y. R. Wang, B. Saiki, A. Anderson, A. Schulz, D. Grossman, and Z. Tatlock, "Rewrite rule inference using equality saturation," *CoRR*, vol. abs/2108.10436, 2021. [Online]. Available: https://arxiv.org/abs/2108.10436