A Survey on Term Rewriting for Code Optimization

Anthony Hunt

March 15, 2025

Contents

1 Introduction **1**

2 Term Rewriting **2**
2.1 The Art of Translation . . . 2
2.2 A (Simple) Term Rewriting System . . . 4
2.2.1 A Small Example of a Simple TRS . . . 5
2.3 Termination . . . 6
2.3.1 Reduction Orders . . . 6
2.3.2 Simplification Orders . . . 6
2.4 Confluence . . . 6
2.5 A Complete Algorithm . . . 7
2.6 Limitations . . . 7

3 An Improved Term Rewriting System for Code Optimization **7**
3.1 Rewriting Strategies . . . 7
3.2 Code Optimization with Rewriting Strategies . . . 7

4 Conclusion **7**

1 Introduction

As a means of attempting to popularize the sheer vastness and absurdity of infinity with regards to information, society has often perpetuated the proverb of monkeys on a typewriter; an infinite number of typewriter-equipped monkeys with an infinite amount of time would eventually produce the complete works of Shakespeare. While the broad sentiment conveyed through this statement can serve as an interesting thought experiment for the masses, computer scientists often have to deal with very real consequences of seemingly infinite swaths of data and computation on finite resources. Indeed, with the advent of generative

AI models and the internet, collecting and distilling temporarily useful information from universal entropy has become a more pressing and arduous task than ever before.

Since computers are precise, powerful machines limited in expressivity only by their need for extremely simple instructions, the subfield of compilers and programming languages is especially concerned with efficient, effective, and provable methods of translation. Term rewriting is one such method that enables compilers to convert very high level language into fast, low-level executable information.

In this paper, we explore concepts and ideas surrounding the topic of simple term rewriting, delving into the inner-workings, benefits, and limitations of such a system. Later, we abstract some components of the simple term rewriting for improved expressivity, control, modularity, and overall usefulness in the context of code optimization.

2 Term Rewriting

2.1 The Art of Translation

Throughout a typical undergraduate program in computer science, a great deal of emphasis is placed on working with text as a form of malleable data. Entire courses are dedicated to the topics of expressions, grammars, programming languages, and discrete mathematics, all of which deeply involve manipulation of textual symbols at both a syntactic and semantic level. And this trend is not without good reason: in the expansive area of predicate logic, purely syntactic manipulation of variables and operators can create hundreds of new theorems from a small set of existing axioms. For example, consider the following axioms that describe addition over natural numbers:

1. $\forall n \in \mathbf{N} : 0 + n = n$
2. $\forall n, m \in \mathbf{N} : Suc(n) + m = Suc(n + m)$

With these two rules, the induction proof style, and some other basic axioms, all well-known properties of addition can be derived through pure syntactic transformation. The commutative property, for example, can be obtained through repeated application of given facts. Before attempting this proof, however, we need to establish the right identity of 0 ($n + 0 = n$):

Proof. Right identity of 0.
Base case: 0 + 0 = 0

$$\begin{aligned} & 0 + 0 \\ =& \langle \text{By axiom 1} \rangle \\ & 0 \end{aligned}$$

Inductive Step (with hypothesis $n + 0 = n$).

$$
\begin{aligned}
&Suc(n) + 0\\
=&\langle \text{By axiom 2} \rangle\\
&Suc(n + 0)\\
=&\langle \text{By induction hypothesis} \rangle\\
&Suc(n)
\end{aligned}
$$

□

With the two axioms and this derived theorem, we can simply manipulate provided terms within the limitations of the given rules to prove commutativity:

Proof. Commutativity of + over addition by induction.
Base case: 0 + n = n + 0

$$
\begin{aligned}
&0 + n\\
=&\langle \text{By axiom 1} \rangle\\
&n\\
=&\langle \text{By right identity of 0} \rangle\\
&n + 0
\end{aligned}
$$

Inductive Step (with hypothesis $n + m = m + n$):

$$
\begin{aligned}
&Suc(n) + m\\
=&\langle \text{By axiom 2} \rangle\\
&Suc(n + m)\\
=&\langle \text{By induction hypothesis} \rangle\\
&Suc(m + n)\\
=&\langle \text{By axiom 2} \rangle\\
&Suc(m) + n
\end{aligned}
$$

□

All this to say that the core of term rewriting is merely an exhaustive application of transformation rules on a piece of text. A term rewriting system (TRS), then, is a system that defines available rules and valid inputs/outputs of those rules. Any valid input/output text is often referred to as a term, with each application of a rule is a rewrite step. A term reaches normal form within a TRS when rules no longer apply to any text within the term (ie. it is irreducible).

In the context of compilers, the goal of a TRS is clear: we aim to translate high level, readable, and expressive text into efficient low-level executable terms. Of course, a TRS cannot hope to shoulder the responsibility of the entire compilation process, but just as a lexer reduces the complexity of a parser, so too can a TRS simplify the generation of useful, efficient code.

Since term rewriting and lambda calculus both have roots in manipulating the structure of symbols as a form of computation, term rewriting systems are uniquely poised to handle symbolic computation (ie. algebra-related mathematics) and functional programming with grace. For instance, formal proofs of correctness for the translation scheme of a TRS follow naturally through enumerating every possible derivation inductively. The function-like nature of rule applications means that, aside from non-determinism and conflicting rules, a developer can guarantee a consistent derivation output.

In most real-world languages that make use of term rewriting, especially the Wolfram Mathematica language, treatment of mathematical expressions as symbolic objects enables both high performance computations and an understandable, step by step derivation of the solution. After all, when attempting to solve a complex expression like $\int\int(\cos x^2 + \tanh x^3)e^y dxdy$, mathematicians naturally make use of well-known formulas and equational logic to translate complex text into something much more manageable.

2.2 A (Simple) Term Rewriting System

In this section, we present a simple term rewriting system as specified in the definitive text by Baader and Nipkow [1].

Formally, a TRS is an abstract reduction system defined by a set of terms T and a relation $\rightarrow: T \times T$ over those terms. Aside from transforming high level expressions into a format digestible for computers, one major goal of this reduction system is to determine whether two (or more) different derivations of a term t will eventually reach the same normal form. In other words, if $t \rightarrow^* a$ and $t \rightarrow^* b$, a and b are joinable to some (eventually normalized) term z (ie. $a \rightarrow^* z$ and $b \rightarrow^* z$ should be true). If a TRS system satisfies this "joinable" property, it is a confluent system. A discussion on the confluence of rewrite systems is contained in Section 2.4.

Additional notation regarding term rewriting systems are listed below, where the details of some properties have been discarded for simplicity and conciseness. These notions will be used intermittently throughout the rest of this paper:

- The signature set $\Sigma = \cup_{n=0}\Sigma^n$ contains function symbols that have arity n (ie. functions that take in n arguments).
- The set of terms $T(\Sigma, X)$ inductively contains all variables X and all applications of functions to those terms.
- The set of ground terms $T(\Sigma, \emptyset)$ contains only constants (ie., $f \in \Sigma^0$) and function applications of those constants. No variables allowed.

- Function $\sigma : X \times T(\Sigma, X)$ is a set of mappings on variables within terms to other elements of $T(\Sigma, X)$.
- A term t is an instance of a term s if a substitution applied to s produces t. Formally, $\sigma(s) = t \implies t \gtrsim s$.

As mentioned in the previous section, term rewriting acts like a translation layer between source text and reduced text. In more formal phrasing, the foundations of term rewriting systems lie in the field of equational logic, where a TRS can draw directional rules from a set of bidirectional ground truths, or identities. The set of identities E contains equations that define which terms are structurally equivalent. For example, $(s \approx t) \in E$ implies that we may swap appearances of s with t and vice versa. An identity $s \approx t$ is valid in E if it is an element within E.

In the induction example from Section 2.1, we made use of identities over natural numbers to derive new identities, refining the relationship between natural numbers and the addition operator. Similarly, term rewriting makes use of transformations and identities to derive new identities. $\sigma(s) \approx \sigma(t)$ is satisfiable in E if there exists a substitution function σ that results in a valid identity. Generally, we aim to create identities that will better direct source text to a correct and consistent normal form.

In general, the word problem of E states that the validity of two terms is undecidable. Without restrictions on the TRS described thus far, it would be impossible to produce a usable, consistent compiler. Even then, if we were to remove variables from the possible set of identities, the word problem (now known as the ground word problem) is still undecidable. The word problem only becomes decidable when we restrict an arbitrary TRS to be terminating and confluent, with a finite amount of rules.

2.2.1 A Small Example of a Simple TRS

To better illustrate the translation-like nature of TRS in the context of equational logic, we implement a very informal proof of commutativity property over natural numbers from the perspective of a TRS.

Consider a set of identities similar to our axioms for natural numbers: $E = \{0 + n \approx 0, Suc(n) + m \approx Suc(n + m), Suc^n(0) \approx n\}$. Then, with a TRS defined by $X = \{m, n\}$ and $\Sigma = \{0, Suc(x_1), x_1 + x_2\}$, we can determine whether a new identity $2 + 3 \approx 3 + 2$ is satisfiable within this set by reducing both sides of the identity to normal form. Note that it is possible to prove $n + m \approx m + n$ in the general case, but for the sake of clarity in one example derivation, we use concrete values.

Starting out, we translate numerical values into their successor function representation and simplify using identities until we have reached normal form.

$$\begin{aligned}
&2+3\\
\approx&\langle\text{By identity E[2]}\rangle\\
&Suc(Suc(0))+Suc(Suc(Suc(0)))\\
\approx&\langle\text{By identity E[1] twice}\rangle\\
&Suc(Suc(0+Suc(Suc(Suc(0)))))\\
\approx&\langle\text{By identity E[0]}\rangle\\
&Suc(Suc(Suc(Suc(Suc(0)))))\\
\approx&\langle\text{By identity E[2]}\rangle\\
&5
\end{aligned}$$

Likewise for $3+2$:

$$\begin{aligned}
&3+2\\
\approx&\langle\text{By identity E[2]}\rangle\\
&Suc(Suc(Suc(0)))+Suc(Suc(0))\\
\approx&\langle\text{By identity E[1] thrice}\rangle\\
&Suc(Suc(Suc(0+Suc(Suc(0)))))\\
\approx&\langle\text{By identity E[0]}\rangle\\
&Suc(Suc(Suc(Suc(Suc(0)))))\\
\approx&\langle\text{By identity E[2]}\rangle\\
&5
\end{aligned}$$

In this example, the exact rewrite rules used correspond with the following substitution function $\sigma = \{0+s \to s, Suc(s)+t \to Suc(s+t), n \to Suc^n(0)\}$. For brevity, we add an additional rule with the condition that it may only be executed in the last step: $Suc^n(0) \to n$. Strictly speaking, this last rule prevents the existence of a normal form and therefore prevents termination of the TRS. The expanded version of 5 is the canonical normal form of the rewrite rules represented by σ since no other reduction rules may be applied. In the case of our simple TRS, we must specify a condition on the last rule that is external to the formal TRS definition itself. Subsequent sections, in addition to Section 3, explore different methods of tackling problems of this nature.

2.3 Termination

2.3.1 Reduction Orders

2.3.2 Simplification Orders

2.4 Confluence

The notion of confluence is essential in building a reliable compiler, since we want to ensure that every compilation execution of the same source text produces consistent, correct, and optimal code.

2.5 A Complete Algorithm

2.6 Limitations

3 An Improved Term Rewriting System for Code Optimization

3.1 Rewriting Strategies

3.2 Code Optimization with Rewriting Strategies

4 Conclusion

References

[1] F. Baader and T. Nipkow. *Term Rewriting and All That*. Term Rewriting and All that. Cambridge University Press, 1998. ISBN: 9780521779203. URL: `https://books.google.ca/books?id=N7BvXVUCQk8C`.

[2] Bruno Buchberger. "Mathematica as a rewrite language". In: *Proceedings of the 2nd Fuji International Workshop on Functional and Logic Programming*. World Scientific. 1996, pp. 1–13.

[3] Thomas J. Watson IBM Research Center, F.E. Allen, and J. Cocke. *A Catalogue of Optimizing Transformations*. IBM Thomas J. Watson Research Center, 1971. URL: `https://books.google.ca/books?id=oeXaZwEACAAJ`.

[4] J. Cooke. *Constructing Correct Software: The Basics*. Formal Approaches to Computing and Inf. Springer, 1998. ISBN: 9783540761563. URL: `https://books.google.ca/books?id=N3whAQAAIAAJ`.

[5] Albert Graf. *The Pure Manual*. 2020. URL: `https://agraef.github.io/pure-docs/pure.html?highlight=term%20rewriting` (visited on 03/14/2025).

[6] Zach Kimberg. *Catln Language Summary*. 2025. URL: `https://catln.dev/` (visited on 03/14/2025).

[7] Mircea Marin and Florina Piroi. "Rule-Based Programming with Mathematica Mircea Marin". In: (May 2004).

[8] Jeremy Miller. *Introduction to Term Rewriting with Meander*. 2023. URL: `https://jimmyhmiller.github.io/meander-rewriting` (visited on 03/14/2025).

[9] *Strategic Rewriting*. 2023. URL: `https://spoofax.dev/background/stratego/strategic-rewriting/strategic-rewriting/` (visited on 03/14/2025).

[10] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. "Building program optimizers with rewriting strategies". In: *SIGPLAN Not.* 34.1 (Sept. 1998), pp. 13–26. ISSN: 0362-1340. DOI: `10.1145/291251.289425`. URL: `https://doi.org/10.1145/291251.289425`.