A Survey on Term Rewriting for Code Optimization

Anthony Hunt

March 16, 2025

Contents

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. Usually, $\rightarrow$ is defined by a mapping of rewrite rules $l \rightarrow r$, which directly translate pieces of text (known as terms) from one form into another. The goal of such a system in the context of compilers is to translate high level, expressive text into a format digestible for computers and efficient in execution.

To ensure the correctness of this reduction process, we must ensure that the system terminates and is confluent:

- Similar to regular executable programs, the termination problem deals with the possibility of reduction derivations of infinite length. We say that a TRS terminates if all possible terms can be evaluated to some irreducible normal form. Therefore, if a TRS does not terminate, there must be an input term t that has an infinite length derivation $t_1 \rightarrow t_2, t_2 \rightarrow t_3, ...$ (ie.

t never finishes the reduction process to a normal form). Further notes on termination can be found in Section 2.3.

- The confluence of a system determines whether two (or more) different derivations of a term t will eventually reach the same normal form. In other words, if $t \to^* a$ and $t \to^* b$, a and b are joinable to some (eventually normalized) term z (ie. $a \to^* z$ and $b \to^* z$ 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.
- The substitution 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 mapping $\sigma = \{0 + s \rightarrow s, Suc(s) + t \rightarrow Suc(s + t), n \rightarrow Suc^n(0)\}$. Because of the direct correlation between rules and identities, building an induction proof for correctness of the the general case, and thus a new identity,

becomes near-trivial. In fact, the proof in 2.1 is more than sufficient, since that style of derivation is essentially equational logic itself.

For brevity, note that 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

Given the striking resemblance between TRS and lambda calculus, the notion of halting becomes an unavoidable problem when attempting to use TRS as a program. In fact, several languages like Catln, Pure, Mathematica, and Meander use term rewriting as the basis of their runtime execution. In other words, the process of reduction in a rewrite system is essentially the same as a recursive functional program. Both computation methods are Turing complete, therefore term rewriting systems must reconcile with the halting problem. However, termination of a TRS is especially important in compilation, since we want to ensure that the compiler itself is able to consistently produce useful output. Further, without a guaranteed normal form for every input (ie. without guaranteed termination for every input), confluence of a TRS is impossible to determine.

Since termination is undecidable for general TRS, we instead examine useful subsets of TRS that enable nearly-as-powerful expressivity but with guarantees of useful output. Some cases of non-terminating TRS are immediately obvious: if the derivation tree contains a cyclic path, the TRS will not terminate. For example, if $l \to r$ where l is a subset of r, the derivation $l \to r, (r[i:j] == l) \to r, (r[i:j][i:j] == l) \to r$ would repeat infinitely.

2.3.1 Reduction Orders

To prove that a rewrite system will terminate, we make use of reduction orders over the set of possible rules. In other words, we aim to give an order to the rules $l_1 \to r_1, l_2 \to r_2, l_3 \to r_3, ...$ such that we can guarantee executing all possible options of rules on a term will eventually result in that term's normal form. Thus, we desire a system that can unequivocally state through transitivity that all terms will start somewhere on the left side of the order $l_1 > l_2, l_2 > l_3, ..., l_i > l_j$, and eventually reach an irreducible form on the right side. Importantly, we note that we do not want to check reduction orders on terms themselves, rather the rules that govern the translation of terms from one form to the next. This prevents a combinatorial explosion in the number of possible reduction pairs.

Formally, a rewrite order $>$ must satisfy:

- Compatibility with functions in Σ. $\forall s_i > s_j : f(s_1, s_2, ..., s_i, ...) > f(s_1, s_2, ..., s_j, ...)$.

- Closure under substitution. $s_i > s_j \implies \sigma(s_i) > \sigma(s_j)$

One example of such an order may be the number of variables in two terms. For example, if s contains 3 occurrences of the same variable and t contains only one occurrence, it is logical that $s > t$, where $>$ compares the number of similar variables in either term, may be a valid and useful reduction order.

A common method of deriving such orders on terms is through an alternative representation in another algebra. For example, if there exists a total substitution function $\sigma : T(\Sigma, X) \times A$ under some algebra A such that $s >_A t \equiv \sigma(s) > \sigma(t)$, the combination of σ and $>_A$ could form a reduction order over all valid rules. Note that an additional requirement of this order is monotonicity, formally that $s_i >_A s_j \implies f(s_1, s_2, ..., s_i, ...) >_A f(s_1, s_2, ..., s_j, ...)$. Generally, the substituted algebraic set consists of polynomials over the natural numbers, a set which can be monotonic and has a clearly irreducible "base" form (ie. 0 cannot be reduced).

2.3.2 Simplification Orders

While reduction orders may be suitable for some simple TRS, they can easily suffer from exponentially large upper bounds on the length of a viable reduction sequence. An informal example of this can be seen by adding exponentiation and multiplication to the above natural numbers example. Depending on the substituted algebraic reduction, the number of Suc calls required for the normal form of 2^{n^2} would be more than exponentially large on the size of n.

As a means of addressing exponential growth in reduction orders, Baader and Nipkow [1] present the concept of a simplification order. A simplification order must satisfy the conditions of a rewrite order, mentioned in Section 2.3.1, but additionally notes that all subterms t_i of a term t may not grow individually, satisfying $t > t_i$. As a consequence, the relative orderings of substitutions into another algebra must be homeomorphic, preserving the relative structure of each subterm.

The text further provides an excellent example for generic automatic termination checking through lexicographic path orders. Semi-formally, $s >_{lpo} t$ iff:

- t is a subterm variable of s, or
- s and t are function applications where $s = f(s_1, ...)$ and $t = g(t_1)$ with one of the following:
  - some subterm $s_i \geq_{lpo} t$
  - $f > g$ and $s >_{lpo} t_j \forall j$
  - $f = g$, $s >_{lpo} t_j \forall j$, and $\exists i : s_i > t_i$

Lexicographic path orders start at the root of a symbol and works towards showing an order on the innermost symbols. This reduction order can be used to show termination of TRS whose reduction sequence length cannot be bounded

by a primitive recursive function. Further, $s >_{lpo} t$ can be decided in polynomial time on t, where termination on the entire TRS is NP-complete.

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. When treating a TRS as if it were executing a program, confluence simply refers to the determinism of the output. Given the same set of inputs, rules, and variables, a TRS that is confluent will always produce the same normal forms on every execution. Note that determinism is not guaranteed by any form of TRS, as any rule with a satisfiable LHS may be applied to a term in any order. If two rules exists with the same LHS but diverging RHS, we must consider the possibility that different rules may be chosen for different compilation executions.

In general, like termination, a finite TRS is powerful enough that confluence is an undecidable problem. However, given knowledge that TRS *is* terminating, we find that confluence is decidable. Furthermore, a terminating TRS is confluent if it is locally confluent. Informally, derivations from t that deviate after one step yet still eventually result in the same normal form are locally confluent.

To calculate whether a pair of diverging rules $l_1 \to r_1$ and $l_2 \to r_2$ are locally confluent from starting term s, we examine three possible cases.

- Case 1: If l_1 and l_2 do not overlap or interfere with one another (ie. they are distinct subterms of s), $s \to^{l_1 \to r_1} t_1 \to^{l_2 \to r_2} z$ and $s \to^{l_2 \to r_2} t_2 \to^{l_1 \to r_1} z$ should result in the same z.

- Case 2: A non-critical overlap of rules where the LHS instance of one rule is properly contained in the substitution function of another rule's LHS. In other words, the term s contains a modified version of l_1 and l_2 such that the most generic forms of both LHS do not overlap. Then, an exhaustive application of both rules where applicable will result in the same normal form.

- Case 3: A critical overlap of l_1 and l_2, where both rules try to manipulate the most generic form of the other's LHS. Then, we define a critical pair as a tuple $(\theta r_1, \theta l_1[\theta r_2]_p)$, where θ is the most general unifier of $l_2 =^? l_1|_p$, which must be examined externally to ensure both transitions result in the same normal form.

If all critical pairs can be resolved to a normal form, we can claim that a TRS is confluent. Of course, since the number of critical pairs is finite, the confluence of a finite terminating TRS is decidable.

Confluence can be determined for some subset of non-terminating rewrite systems, but the topic is beyond the scope of this paper.

2.5 A Complete Algorithm

Now that we have addressed the two major concerns of term rewriting systems, we can derive a generic algorithm, or strategy, to actually process the translation information embedded into rewrite rules. Given the proof of termination through the reduction orders listed in Section 2.3 and a list of identities E, demonstrating confluence of a TRS primarily involves constructing all possible critical pairs from the list of identities and adding resolution rules for the normalized versions of those pairs according to the given reduction order.

Pseudocode for this algorithm is as follows:

Algorithm 1 Basic Completion Algorithm

Require: E: set of identities, $>$: reduction order over $T(\Sigma, X)$
Ensure: Output is either a TRS $R \equiv E$ or failure
  if $(s \approx t) \in E$ cannot be ordered by $>$ **then** return failure
  end if
  $i := 1; R_0 := \{l \to r | (l \approx r) \in E \wedge l > r\}$
  while true **do**
    $R_i := R_{i-1}$
    for all critical pairs s, t **do**
      $s_1, t_1 :=$ normal forms of s, t
      if $s_1 \neq t_1$ and those cannot be ordered by $>$ **then** return failure
      end if
      $R_i := R_i \cup \{l \to r | l, r \in \{s_1, t_1\} \wedge l > r\}$
    end for
    if $R_{i-1} = R_i$ **then** return R_i
    end if
    $i := i + 1$
  end while

As an aside, the resolution process of critical pairs can be conceptualized as resolving merge conflicts between diverging rules. In a sense, two diverging rules are akin to two developers attempting to modify the same line of code. Git, of course, requires manual intervention to minimize the chance of disruptions upon code execution, but the eventual resolution of diverging lines is similar to the process of this algorithm. The goal of the algorithm can therefore be rephrased as pre-emptively generating canonical resolutions for all possible merge conflicts caused by critical pairs.

The limitations of this procedure are fairly standard: the program will fail if terms cannot be ordered, or it may attempt to infinitely create new rules. Further, when successful, the algorithm creates a large number of rules to resolve each critical pair along its derivation path, resulting in long runtimes and high space complexity.

Improvements to this algorithm primarily involve dealing with the TRS and identity set on a higher level than previously seen, where a completion procedure repeatedly follows a set of generic inference rules, described below:

- Deduce: If $l \to r_1, l \to r_2 \in R$, add $r_1 \approx r_2$ to E.
- Orient: If $(s \approx t) \in E$ and $s > t$, add $s \to t$ to R.
- Delete: Remove reflexive identities $s \approx s$.
- Simplify-identity: Transform identity $s \approx t$ to $u \approx t$ if $s \to u \in R$.
- R-simplify-rule: Transform rule $s \to t$ to $s \to u$ if $t \to u \in R$.
- L-simplify-rule: Convert rule $s \to t$ to identity $u \approx t$ if $s \to^{\supset} u$ (s is the more generic form of u).

The behaviour of the combined ruleset essentially shifts the balance of rule derivations and identities between one another. For example, deduce adds an identity to E if the same source term provides two different derived terms in R, covering all critical pairs. Likewise, new rules are derived from identities through the orient rule. Transitive closures are then covered by the simplify series of rules, resulting in a smaller but still correct rule set R.

2.6 Limitations

Although term rewriting systems have a natural affinity for formal proofs of correctness and a straightforward implementation through repeated rule application, the solvers presented thus far have several notable limitations that grossly limit practical use of the presented TRS in mainstream languages.

When designing a TRS for use in a compiler, developers must consider restrictions on NP-complete algorithms for generic termination-related reduction orders, large memory requirements for exhaustive confluence search, and unintelligent application of rules to generate every possible normal-form transformation. Further, while rewrite systems lend themselves well to functional, pattern-matching based languages like Mathematica, ML, and Haskell, the validity of rules becomes extremely difficult to determine when considering side effects or imperative style languages. Even within purely discrete mathematical computations, common identities like commutativity and associativity of binary operators cannot be directly accepted, lest the language cease to terminate. Intuitively, term rewriting works best when there is a straightforward, simplified form for every possible expression.

Often times, to accommodate termination and confluence limitations while attempting to satisfy requirements, the implementation of a TRS will contain external directives intended to circumvent particular instances of non-terminating rules or to encode additional knowledge for more efficient methods of term resolution [9]. However, this process is unmaintainable and tightly coupled to a specific domain or language. By tying down the theoretical formulation of a TRS to its implementation, we lose reusability and modularity.

In some cases, it may even be desirable to combine two TRS together. Of course, doing so presents a fair set of theoretical challenges involving termination

and confluence, as discussed in Chapter 9 of [1], but attempting to combine non-standard implementations of two TRS is futile from an engineering perspective.

Section 3 presents a revised term rewriting system to address these issues.

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.