A Survey on Term Rewriting for Code Optimization

Anthony Hunt

March 20, 2025

Contents

1 Introduction

Before personal computing became synonymous with modern life, conceptualizing the vast potential of decision making machinery was likened to a proverb of monkeys on typewriters; an infinite number of typewriter-equipped monkeys with an infinite amount of time would eventually produce the complete works of Shakespeare. While the conveyed sentiments of infinite randomness 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 non-deterministic computation on finite resources.

By design, computers are precise, powerful machines limited only by their extremely inexpressive low-level interface. Therefore, the prime directive of programming languages and compilers aims to provide a more intuitive and expressive experience when communicating with machines, while ensuring preservation of correctness and highly efficient translated instructions. Formal, provable methods are vital to guarantee that real-world compilers meet these goals.

In this paper, we explore concepts and ideas surrounding the topic of simple term rewriting, a translation scheme that offers both a traceable and understandable foundation for compiler development. First, we delve into the innerworkings, benefits, and limitations of rewriting systems. Later, we abstract some components of 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 the natural numbers 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 as a rewrite step. A term reaches normal form within a TRS when rules no longer apply to any text within the term (i.e. 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 (i.e. algebra-related mathematics) and functional programming with grace [1]. 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 [3, 10]. After all, when attempting to solve a complex expression like $\int \int (\cos x^2 + \tanh x^3) e^y dx dy$, 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 $t_1 \rightarrow t_2, t_2 \rightarrow t_3, ...$ (i.e.

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 $t \rightarrow^* a$ and $t \rightarrow^* b$, a and b are joinable to some (eventually normalized) term z (i.e. $a \rightarrow^* z$ 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 is 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 (i.e. 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. Generically, terms follow the grammar $t ::= x|c|f(t_1, ..., t_n)$ where x is a variable in X, c is a constant in Σ^0, and f is a function in Σ^n.
- The set of ground terms $T(\Sigma, \emptyset)$ contains only constants (i.e., $f \in \Sigma^0$) 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:

$$E = \{0 + n \approx 0, Suc(n) + m \approx Suc(n + m), Suc^c(0) \approx c\}$$

Then, with a TRS defined by $X = \{m, n\}$ and $\Sigma = \{c, 0, Suc(x_1), x_1 + x_2\}$, 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:

$$\sigma = \{0 + s \to s, Suc(s) + t \to Suc(s + t), c \to Suc^c(0)\}$$

Because of the direct correlation between rules and identities, building an induction proof for correctness of the general case, and thus a new identity, becomes near-trivial. In fact, the proof in Section 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: $Suc^c(0) \to c$. Strictly speaking, this last rule blocks all terms from reaching a normal form, thereby preventing 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 that require more control over the derivation process.

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 [8, 6, 10, 11]. 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 (i.e. without guaranteed termination for every input), determining confluence of a TRS becomes very difficult.

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.

Deriving orders on terms is commonly achievable through alternative representation of terms in another algebra. 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 (i.e. 0 cannot be reduced).

Consider a set of rules R that mirrors the example TRS for addition over natural numbers defined in Section 2.2.1:

$$R = \{0 + n \to n, n + 0 \to n, Suc(n) + m \to Suc(n + m), c \to Suc^c(0)\}$$

Note that we explicitly exclude the inverse of $Suc(n) + m \to Suc(n + m)$ since such a rule would enable commutativity and thus prevent termination of the TRS. Additionally, we denote the set of possible terms T as all combinations of $\{c, x_1 + x_2, Suc(x_1)\}$ where x_i denotes variables and c is a constant $\in \mathbb{N}$. Informally, this system will reduce every combination of numbers under the $+$ symbol to a single number defined by successive Suc constructors. To prove termination of the system, we map every possible term, defined by combinations of numbers, Suc constructors, and addition operators in the following manner:

Operator	Polynomial Reduction
$c \in \mathbb{N}$	$c + 1$
$Suc(x_1)$	$P(x_1) + 1$
$x_1 + x_2$	$(P(x_1) + P(x_2))^2$

Table 1: Polynomial equivalents for a reduction of addition over $\mathbb{N}$

For each possible term combination as examined by the rules in R, we observe the following:

- $0 + n \rightarrow n$ and $n + 0 \rightarrow n$ will always decrease the value of the equivalent polynomial reduction. The equivalent polynomials of the LHS are $P_+(P(0), P(n)) = P_+(1, X) = (X + 1)^2$, where X is the unknown polynomial representation of a term n. Since the RHS is always $P(n) = X$ and $(X + 1)^2 > X$, these two rules satisfy the polynomial reduction.

- $c \rightarrow Suc^c(0)$ converts the numerical representation of a constant into repeated Suc constructions. Because we defined $P(c) = c+1$ and $P_{Suc}(0)^c = 0 + 1 + 1 + ... = \sum_0^c 1 = c$, this rule satisfies LHS > RHS.

- $Suc(n) + m \rightarrow Suc(n + m)$ moves addition inside the Suc constructor. The LHS representation is $P_+(P_{Suc}(P(n)), P(m)) = ((X + 1) + Y)^2$ and RHS is $P_{Suc}(P_+(P(n), P(m))) = (X + Y)^2 + 1$, where X, Y correspond to $P(n), P(m)$ respectively. Expanding both sides, we find that $X^2 + Y^2 + 2XY + 2X + 2Y + 1 > X^2 + Y^2 + 2XY + 1$, so LHS > RHS.

Since all possible rule reductions decrease monotonically over the polynomial algebra, we can guarantee termination of this TRS.

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 a term like 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 guideline for generic automatic termination checking through lexicographic path orders. Semi-formally, $s >_{lpo} t$ iff:

- Condition 1: t is a subterm variable of s, or

- Condition 2: s and t are function applications where $s = f(s_1, ...)$ and $t = g(t_1, ...)$ with one of the following:

  - Condition 2.1: some subterm $s_i \geq_{lpo} t$
  - Condition 2.2: $f > g$ and $s >_{lpo} t_j \forall j$
  - Condition 2.3: $f = g$, $s >_{lpo} t_j \forall j$, and $\exists i : s_i > t_i$

Lexicographic path orders start at the root of a symbol and work 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.

For completeness, we show that the addition TRS described above is a simplification order over $>_{lpo}$. Referring to the set

$$R = \{0 + n \to n, n + 0 \to n, Suc(n) + m \to Suc(n + m), c \to Suc^c(0)\}$$

we show:

- $0 + n \to n, n + 0 \to n$ are satisfied by the first condition. n is a subterm of both $n + 0$ and $0 + n$.
- In the rule $c \to Suc^c(0)$, c can be considered an 0-ary constant. Since the definition of $>_{lpo}$ expects an ordering of functions as input, the 0-ary function $c > Suc$, and c is certainly greater than the subterm 0. Thus, condition 2.2 is satisfied.
- $Suc(n) + m \to Suc(n + m)$ is again function applications more clearly of the form $+(Suc(n), m) \to Suc(+(n, m))$. We can say that $+ > Suc$ to follow condition 2.2. As part of 2.2, we must check that $+(Suc(n), m) >_{lpo} +(n, m)$. Turning to condition 2.3, we see that all subterms of the LHS are greater than the RHS ($+(Suc(n), m) >_{lpo} n$ and $+(Suc(n), m) >_{lpo} m$ by condition 1), and there exists a subterm of the LHS > RHS ($Suc(n) >_{lpo} n$ by condition 1).

Although the proof for termination seems more complex than the polynomial scheme in Section 2.2, the simple generality of recursively following conditions enables fast derivations of such proofs.

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 exist with the same LHS but diverging RHS, we must consider the possibility that different rules may be chosen for different instances of compilation execution.

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 (i.e. they are distinct subterms of s), $s \xrightarrow{l_1 \to r_1} t_1 \xrightarrow{l_2 \to r_2} z$ and $s \xrightarrow{l_2 \to r_2} t_2 \xrightarrow{l_1 \to r_1} z$ 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. Moreover, 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. We first assume that diverging paths will eventually reduce to the same normal form. Then, if we find that two paths never converge, we know that the TRS is not confluent. 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 [17]. 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 [14]. 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 limitations.

3 An Improved Term Rewriting System for Code Optimization

Modularity and abstraction are key pillars in supporting the boundless complexity of modern day software engineering. In the case of a generic TRS solver, limited modularity is achieved through direct modification of the available ruleset. Unfortunately, the one-size-fits-all design relegates domain-specific knowledge from the theoretical, provable level to the implementation level, restricting the flexibility and efficiency of real-world applications.

Therefore, we now describe a truly modular system that can take full advantage of domain specific knowledge and conditional rule following. Instead of blindly choosing reduction rules in an attempt to eventually reach normalized terms, we can strategically choose to apply specific rules in contexts where we know more than a simple TRS. Algorithms made from this higher-level formalism, known as strategies, provide a layer of insulating abstraction between the purely theoretical world of TRS and other domains whose knowledge may prove useful in the implementation of a compiler. In simpler terms, strategies enable the creation of algorithms customizable to a domain but external to the term rewriting process itself.

Notions of termination and confluence are still troublesome to some degree, as with any TRS, but abstracted strategies can derive generic solutions for critical pairs based on context at the theoretical level rather than the implementation level. Further, a combinatorial explosion of possible rules can be elegantly mitigated through intelligent decisions made in the strategy algorithm.

3.1 Rewriting Strategies

A rewrite strategy system extends a TRS by assigning labels $L : l \to r$ to each rule $l \to r$ and providing a set of operators on those labels. The user defined strategy is an algorithmic application of labelled rules to achieve a targeted normal form. Since we no longer apply all rules indiscriminately, we must take care to ensure that the desired normal forms are derivable by both available rules and application of rules within the strategy. As an additional note, strategies can be compiled with higher-level strategies to provide optimizations not only on the intended target source code, but bootstrapped for the compiler itself.

The default TRS strategy can be modelled by a directed graph with branches following all possible paths of derivations. Formally, the graph contains the recursive transitive closure between all valid rule orderings. Customized strategies trim down the size of this reduction space by cutting off undesirable paths using contextual knowledge of the current reduction state.

For simplicity, we reuse the same notation as in the previous TRS, with the addition of a few rudimentary operators and properties of strategies:

- Terms can additionally contain tuples: $t ::= t|(t_1, ..., t_n)$. This is essentially syntactic sugar for an id function with arity n, but useful nonetheless.

- A strategy $t \xrightarrow{L:l \to r} s$ that uses a rule L succeeds if there exists a substitution that applied to l, r produces t, s. In other words, the rule is valid if some subterm of t and s match to l and r respectively. The strategy fails with result $\uparrow$ if no such substitution exists. Strategies as described by [17] are often defined through typical functional syntax, so $\texttt{id}(s_1) = s_1$ would be a strategy with similar behaviour to Haskell's own `id` function. Although we overload the $\to$ operator between rules and strategies, at a most basic level, a rule can be considered a strategy if desired. For the rest of this section, $\to$ denotes the more generic version of the relation, accommodating both rules and strategies alike.

- Denote the set of strategies S as all possible relations defined by $\to$. The non-deterministic choice operator $+ : S \times S \to T \cup \uparrow$ then accepts two rules and succeeds with a result if at least one rule is valid. Note that backtracking for this operator is only surface level, so for a strategy $s_1 + s_2$, if s_1 is non-deterministically selected and fails immediately, s_2 is then selected. However, if s_1 only fails after a long series of internal derivations, the entire strategy $s_1 + s_2$ will fail.

- The deterministic left-choice operator $\nleftarrow : S \times S \to T \cup \uparrow$ prefers to follow the leftmost strategy, only attempting the rightmost strategy upon failure of the left. Similar to $+$, backtracking for failures is only surface-level.

- The composition of strategies $; : S \times S \to T \cup \uparrow$ will apply the first strategy and then the second strategy sequentially. Both input strategies must succeed.

- The recursion operator $\mu x : S \to T \cup \uparrow$ is slightly different from the usual operator syntax. Strategies defined by μx have the form $\mu x(s)$, where a recursive call only happens where x appears within s. For example, the strategy function $\texttt{repeat}(s) = \mu x((s; x) \nleftarrow \epsilon)$ will recursively apply the strategy s on all subterms of a term until failure. Then, since the other argument ϵ will always succeed with no additional action, the strategy `repeat` will finally terminate.

Given the notation defined here, there is no way to distinguish between a strategy that runs on the root node or one that runs on a child node. However, this separation of valid strategy applications is important in maintaining granular control of that strategy's actions. There may be some cases where we wish to apply a rule to only one subterm of the current term, or restrict applications to only the root term. Therefore, we define a special strategy $i(s)$ is that executes the strategy s on the ith child of the current term. This function enables the segregation of strategies only applicable at the root of a term and those that should be applied only on a particular child.

Further useful strategies for handling child subterms are defined below:

- $\Box(s)$ applies s to all children and only succeeds if all applications of s succeed. Intuitively, this acts like a loop or `map` function, running s on every element of a term.

- $\diamond(s)$ non-deterministically applies s to one child, only failing if all children fail. Similar to $\Box$, an implementation may loop over the list of randomized subterms, returning immediately on the first success.

- $\boxtimes(s)$, like $\Box$, applies s to all children. However, like $\diamond$, this operation only fails if all children fail.

With these available strategies, we can now formulate ultra-concise algorithms to represent three typical methods for evaluating generic TRS:

- $\texttt{reduce}(s) = \texttt{repeat}(\mu x(\diamond(x) + s))$, which applies a strategy s (or in this case, the entire set of TRS joined with $+$) repeatedly somewhere within the starting term.

- $\texttt{outermost}(s) = \texttt{repeat}(\mu x(s \nleftarrow \diamond(x)))$, a top down evaluation approach starting from the outermost structure of the root term.

- $\texttt{innermost}(s) = \texttt{repeat}(\mu x(\diamond(x) \nleftarrow s))$, a bottom up approach that acts on the leaf children of a term. Although this definition is inefficient since it must search the entire tree for root nodes for every application of s, an alternative strategy is easily constructed through application of the recursive function to all subterms in the term tree before evaluation.

As is evident in the conciseness of `reduce` compared to Section 2.5's Basic Completion Algorithm, the rewrite strategy system defined here is noticeably more expressive and understandable. Nevertheless, this system still lacks the context-awareness required to handle some rules like commutativity and associativity. In the next section, we specify additional components of rewrite strategies that enable full contextual awareness and controllability of the strategy program.

3.1.1 Further Improvements on Rewrite Strategies

To accommodate the necessity of side effects and contexts in devising strategies for a programming language optimizer, Visser et al [17] proposes a method of breaking down individual rules into atomic pieces: `match` and `build`. Internally, the rule application for $l \rightarrow r$ on a term t first binds the variables of l to matching subterms in t, then builds a new term by replacing the bound subterms of t with the rightside transformations specified by r. Formally, $l \rightarrow r$ now becomes the sequence $\texttt{match}(l)$; $\texttt{build}(r)$.

Notions of environments and scopes are introduced to formalize the transfer of bindings from `match` to `build` over the ; operator. An environment $\mathcal{E}$ contains a mapping of variables to ground terms, where the `match` operation applied to a term t records the variable binding and preserves it for use in `build`. Since variable names may be reused in different contexts, a scope surrounding a strategy s may temporarily redefine a list of variables for use only within the strategy. Conceptually, scopes are mini-environments which contain local variables used within a strategy. Formally, scopes are demarcated by $\{\vec{x} : s\}$,

where $\vec{x}$ is a list of variables defined only within the expression encased by brackets.

Making use of these environments, `where` is introduced as a function to allow an additional strategy to operate on variables within a rule. For example, the rule $L : l \to r$ `where` s is equivalent to $L = \{$variables of l, r, s : `match`(l); `where`(s); `build`$(r)\}$. `where` does not directly modify the inputs of l, but can change the values of bound variables of l based on the total environment and input of l. In turn, this effectively carries only the side effects of s, rather than a direct transformation. `where` is also useful for placing conditional restrictions on rule evaluation, as seen in Section 3.1.2.

Finally, we can build context aware systems capable of detecting cycles in the derivation graph, make decisions based on environmental knowledge, and break down the essence of term rewriting into granular pieces. Further syntactic sugar is provided by Visser et al [17] to simplify the process of defining non-linear rules, boolean condition testing, and using strategies within rules themselves. The development of an optimizing compiler using term rewriting is within reach with the foundational correctness of theoretical, formal proofs.

3.1.2 A Small Sample Strategy

Earlier in this text, we claimed that rewrite strategies (along with environments and scopes) provided a more powerful infrastructure for describing translations that would otherwise fail in a simple TRS. For instance, commutativity over binary operators or conversions between constructor and numerical form were previously unattainable or only used in extraneous circumstances. Allowing either of these rules within a simple TRS would prevent the system from terminating and sully any attempts to decide confluence. Now, we extend the example of addition over natural numbers to include both commutativity rules and transformations between normal forms.

While this may be an abuse of the environment system described in [17], a simple and effective way of preventing infinite derivation chains caused by the commutativity identity $m + n \approx n + m$ is to monitor all derivations of the commutativity rule. Then, once a commutativity rule is followed in one direction for a scoped set of terms, we simply prevent that rule from being followed in the opposing direction. Lenient pseudocode for this strategy would look something like:

$$
\begin{aligned}
&\texttt{CommuteRule} = \{ \\
&\quad vars(x_1,x_2) : \\
&\qquad\qquad match(x_1 + x_2); \\
&\qquad\qquad test((x_2, x_1) \notin V); \\
&\qquad\qquad where(V := V \cup \{(x_1, x_2)\}); \\
&\qquad\qquad build(x_2 + x_1) \\
&\qquad\qquad \}
\end{aligned}
$$

First, we bind the subterms x_1, x_2 of a $+$ term to the local scope (note that $x_1 + x_2$ here is a pattern matching of the addition operator, rather than the non-deterministic choice of two strategies). Then, we test that the inverse commutation of the bound subterms does not appear in some global set V of followed commutative pairs. After the test, we save the currently seen pair to the global set V as a side effect and eventually build back the commuted version of the input. This method could certainly be improved with scoping to enable inverse commutation operations in more environments, especially if variable names have been reused, but the goal of this example is to demonstrate the additional flexibility and control offered by a formal rewrite strategy. A rule like this one may be applied like a normal TRS rule, but will deliberately fail more often than its unconditional counterpart to directly address non-terminating behaviour.

In a previous example, we made use of the identity $Suc^c(0) \approx c$ to convert a constructor-based value into an understandable numerical value. However, we made the rule an exceptional case in that it should only be converted once all $+$ operators have been removed from the complete term (i.e. it can only run at the end of the derivation). A rewrite strategy with equivalent behaviour would be as follows:

$$
\begin{aligned}
&\texttt{in} = \{ \\
&\quad x, y : \\
&\qquad match(x); \\
&\qquad match(y); \\
&\qquad test(build(y); \mu x(match(x) \Leftarrow \diamond(x))) \\
&\quad \}
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{rootPrettyPrint} = \{ \\
&\quad t : \\
&\qquad match(t); \\
&\qquad where(\neg \texttt{in}(x_1 + x_2, t)); \\
&\qquad build(t) \\
&\quad \}
\end{aligned}
$$

The strategy `rootPrettyPrint` simply checks for terms of structure $x_1 + x_2$ within the bound input term t. If a $+$ operator is found, the call to *where* will fail and the entire rule will cease to execute. The `in` strategy, provided by [17], binds two terms, and recursively attempts to match occurrences of $+$ from within t. When a match is found, the strategy returns successfully. Through this strategy, we have effectively encoded into our rewrite strategy, without modifying the underlying TRS, that rendering a constructor-based value back to numerical form can only happen after all calculations involving addition are complete (i.e. all non-aesthetic computation has finished).

3.2 Functional Code Optimization with Rewriting Strategies

Although the field of code optimization and optimal code generation is as vast as the number of languages themselves, we can observe a few objective optimizations available for functional programs made simple by rewriting strategies. For example, dead code can be removed by extracting terms without side effects from unused subexpressions. Consider the expression `let x = 1 in z`. By using `where` to test for occurrences of x in z and assess any side effects of x, we can conditionally perform a rewrite action to trim the unused `let` portion of the statement. The optimizer gains the ability to use syntactic analysis side-by-side with semantics through modified environment sets.

Further code optimizations like function inlining and replacing occurrences of variables with respective values are made possible by context-aware rules and strategies. For instance, in function inlining, we must ensure simultaneous substitution of all occurrences of a function within some set of terms. Then, the optimizations of the function itself must be applied to all replaced occurrences. With a generic fully non-deterministic rewrite system, guaranteeing performant inlining of functions would not be possible, since any rule could be applied at any time. Moreover, unwrapping loops and inlining variables used within those loops is an easy performance improvement only possible with knowledge of the variable's lifetime. If a strategy finds that a variable only exists as an iterator for a loop, unused elsewhere in the code, that variable may be inlined. Additionally, such a strategy may also detect other variables within the loop that are the result of functions applied to iterator itself. In that case, the variables may be replaced with appropriate constants.

An extensive thesis by Paraskevopoulou [12] contains a full catalogue of function optimizations, along with rewrite strategies to perform those optimizations. For brevity, we mention only a few examples of particularly interesting optimizations. The thesis denotes strategies to combine nested function un-currying with inlining rules to massively reduce the number of required function calls in generated (normal form) code. Function parameters may be removed if unused, with similar checks to dead code elimination. In the same vein, lambda lifting to promote locally defined functions to the global scope can be used in conjunction with unnesting closure conversions to minimize inter-function calls and dependencies.

Interestingly, the Glasgow Haskell Compiler makes use of similar rewrite-based optimizations during the process of code simplification [2]. The simplification process, which happens in between a series of other optimization programs, implements a strategic TRS not unlike the one described in section 3.1 . Rewrite rules specified in a miniature functional language `Core` can directly extend the compiler's simplification component, granting the ability to add domain-specific optimizations for certain functions. A developmental Haskell paper [13] even describes a system for automatically deriving rewrite rules based on static code analysis. However, user-specified rules must be used with caution, since GHC makes no attempts to verify the correctness or soundness of such rules.

4 Conclusion

Term rewriting systems are a vital concept to producing simple, formally verifiable compilers. Throughout this paper and even in the real world, the idea of a simple TRS consisting of rules and terms has permeated nearly every aspect of modern functional and symbolic programming. The subfield of rewriting has been modified, extended, and decomposed countless times all in extremely unique ways. Naturally, each flavour of term rewriting has its advantages and disadvantages, but the pervasiveness of the original idea is a testament to the power of a generic set of transformations.

Although rewriting strategies are one among many refined translation techniques, the practical and useful separation of rules and rule applications is as instinctive as separating the backend and frontend of a website. Either component of the system can be extended at any time with minimal impact on the other, just as a server may host additional APIs or a website implement aesthetic changes with complete inter-component isolation. In essence, rules of a TRS provide atomic, provable, and correctness-preserving instructions, while the rewrite strategy governs an otherwise indiscriminate rule execution into an efficient, practical, controllable system. Granting even a gentle push in the right direction can massively reduce the number of possible derivation trees for a TRS, mitigating issues caused by non-determinism. Yet, the powerful nature of random pattern matching remains an accessible tool.

The rewrite system presented in this paper aims to harness the power of non-determinism while remaining in control of the seemingly uncontrollable, predictable in the face of unpredictability, and most importantly, correct in the presence of infinite possibility. Computation systems have reached a level of theoretical power that, given enough time and space, could generate any and every possible combination of data. The hurdle that we face now, then, is cleverly sieving through nonsensical information to recognize optimal patterns in the mundane. Syntax based tools in line with rewriting, grammars, and automata are the only realistic methods in which we can formalize the concept of language and symbolic interaction into something digestible and usable.

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] A. Brown and G. Wilson. *The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks*. The Achrictecture of Open Source Applications. Creative Commons, 2011. ISBN: 9781257638017. URL: https://books.google.ca/books?id=pgI1AwAAQBAJ.

[3] 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.

[4] 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.

[5] 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.

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

[7] Mark Hills et al. "A case of visitor versus interpreter pattern". In: *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer. 2011, pp. 228–243.

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

[9] Cynthia Kop and Naoki Nishida. "Term rewriting with logical constraints". In: *International Symposium on Frontiers of Combining Systems*. Springer. 2013, pp. 343–358.

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

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

[12] Zoe Paraskevopoulou. *Verified Optimizations for Functional Languages*. Princeton University, 2020.

[13] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. "Playing by the rules: rewriting as a practical optimisation technique in GHC". In: *2001 Haskell Workshop*. ACM SIGPLAN. Sept. 2001. URL: https://www.microsoft.com/en-us/research/publication/playing-by-the-rules-rewriting-as-a-practical-optimisation-technique-in-ghc/.

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

[15] Yoshihito Toyama. "Commutativity of term rewriting systems". In: *Programming of future generation computers II* (1988), pp. 393–407.

[16] Mark GJ Van Den Brand, Paul Klint, and Jurgen J Vinju. "Term rewriting with traversal functions". In: *ACM Transactions on software engineering and methodology (TOSEM)* 12.2 (2003), pp. 152–190.

[17] 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.