When cooking it is very important to follow the rules. You don't need to stick to an exact recipe, but you do need to know the how ingredients will react to temperature and how different combinations will taste. Otherwise you might get some unexpected reactions.

Similarly, there isn't a single way to compile Curry programs, however we do need to know the rules of the game. Throughout this compiler, I'll be transforming Curry programs in many different ways, and it's important to make sure that all of these transformations respect the rules of Curry.

As we'll see, If we break these rules, then we may get some unexpected results.

1 Rewriting

In programing language terms, the rules of Curry are its semantics. The semantics of Curry are generally given in terms of rewriting. [**?**] While there are other semantics [**?**], rewriting is a good fit for Curry. We'll give a definition of rewrite systems, then we'll look at two distinct types of rewrite systems. Term Rewrite Systems, which is used to implement transformations and optimizations on the Curry syntax trees; and Graph Rewrite Systems, which defines the operational semantics for Curry programs. This will be mathematical foundation will help us justify the correctness of our transformations even in the presence of laziness, non-determinism, and free variables.

An Abstract Rewrite System (ARS) is a set A along with a relation $\to$. we write $a \to b$ instead of $(a, b) \in \to$, and we have several modifiers on our relation.

- $a \to^n b$ iff $a = x_0 \to x_1 \to \ldots x_n = b$.
- $a \to^{\leq n} b$ iff $a \to^i b$ and $i \leq n$.
- reflexive closure: $a \to^= b$ iff $a = b$ or $a \to b$.
- symmetric closure: $a \leftrightarrow b$ iff $a \to b$ or $b \to a$.
- transitive closure: $a \to^+ b$ iff $\exists n \in \mathbb{N}. a \to^n b$.
- reflexive transitive closure: $a \to^* b$ iff $a \to^= b$ or $a \to^+ b$.
- rewrite derivation: a sequence of rewrite steps $a_0 \to a_1 \to \ldots a_n$.
- a is in *normal form* if no rewrite rules can apply.

A rewrite system is meant to invoke the feeling of algebra. In fact, rewrite system are much more general, but the can still retain the feeling. If we have an expression $(x \cdot x + 1)(2 + x)$, we might reduce this with the reduction in figure **??**.

We can conclude that $(x \cdot x + 1)(x + 2) \to^+ x^3 + 2x^2 + x + 2$. This idea of rewriting invokes the feel of algebraic rules. The mechanical process by rewriting allows for a straightforward implementation on a computer. Therefore, it

	$(x \cdot x + 1)(2 + x)$	
$\rightarrow$	$(x \cdot x + 1)(x + 2)$	by commutativity of addition
$\rightarrow$	$(x^2 + 1)(x + 2)$	by definition of x^2
$\rightarrow$	$x^2 \cdot x + 2 \cdot x^2 + 1 \cdot x + 1 \cdot 2$	by FOIL
$\rightarrow$	$x^2 \cdot x + 2x^2 + x + 2$	by identity of multiplication
$\rightarrow$	$x^3 + 2x^2 + x + 2$	by definition of x^3

Figure 1: reducing $(x \cdot x + 1)(2 + x)$ using the standard rules of algebra

shouldn't be surprising that most systems have a straightforward translation to rewrite systems.

It's worth understanding the properties and limitations of these rewrite systems. Traditionally there are two important questions to answer about any rewrite system. Is it *confluent*? Is it *terminating*?

A confluent system is a system where the order of the rewrites doesn't change the final result. For example consider the distributive rule. When evaluating $3 \cdot (4 + 5)$ we could either evaluate the addition or multiplication first. Both of these reductions arrived at the same answer as can be seen in figure **??**.

	$3 \cdot (4 + 5)$
$\rightarrow$	$3 \cdot 4 + 3 \cdot 5$
$\rightarrow$	$12 + 15$
$\rightarrow$	27

(a) distributing first

	$3 \cdot (4 + 5)$
$\rightarrow$	$3 \cdot 9$
$\rightarrow$	27

(b) reducing $4 + 5$ first

Figure 2: Two possible reduction of $3 \cdot (4 + 5)$. Because they both can rewrite to 27, this is a confluent system.

A terminating system will always halt. That means that eventually there are no rules that can be applied. Distributivity is terminating, where as commutativity is not terminating. See figure **??**.

	$a \cdot (b + c)$
$\rightarrow$	$a \cdot b + a \cdot c$

	$x + y$
$\rightarrow$	$y + x$
$\rightarrow$	$x + y$
$\ldots$	

Figure 3: A system with a single rule for distribution is terminating, but any system with a commutative rule is not. Note that $x + y \rightarrow^2$ x + y

Confluence and termination are important topics in rewriting, but we will largely ignore them. After all, Curry is neither confluent nor terminating. However, there will be a few cases where these concepts will be important. For example, if our optimizer isn't terminating, then we'll never actually compile a

program.

Now that we have a general notation for rewriting, we can introduce two important rewriting frameworks. Term Rewriting and Graph Rewriting, where we are transforming trees and graphs respectively.

1.1 Term Rewriting

As mentioned previously, the purpose of term rewriting is to transform trees. This will be useful in optimizing the abstract syntax trees (ASTs) of Curry programs. Term Rewriting is a special case of Abstract Rewriting. Therefore everything from abstract rewriting will apply to term rewriting.

A term is made up of signatures and variables. We let Σ and V be two arbitrary alphabets, but we require that V be countably infinite, and $\Sigma \cap V = \emptyset$ to avoid name conflicts. A *signature* $f^{(n)}$ consists of a name $f \in \Sigma$ and an arity $n \in \mathbb{N}$. A *variable* $v \in V$ is just a name. Finally a *term* is defined inductively. The term t is either a variable v, or it's a signature $f^{(n)}$ with children $t_1, t_2, \ldots t_n$, where $t_1, t_2, \ldots t_n$ are all terms. We write the set of terms all as $T(\Sigma, V)$.

If $t \in T(\Sigma, V)$ then we write $Var(t)$ to denote the set of variables in t. By definition $Var(t) \subseteq V$. We say that a term is linear if no variable appears twice in the term.

This inductive definition gives us a tree structure for terms. As an example consider Piano arithmetic $\Sigma = \{+^2, *^2, -^2, <^2, 0^0, S^1, \top^0, \bot^0\}$. We can define the term $*(+(0, S(0)), +(S(0), 0))$. This gives us the tree in figure **??**. Every term can be converted into a tree like this and vice versa. The symbol at the top of the tree is called the root of the term.

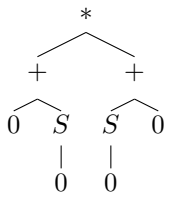


Figure 4: Tree representation of the term $*(+(0, S(0)), +(S(0), 0))$.

A *child* c of term $t = f(t_1, t_2, \ldots t_n)$ is one of $t_1, t_2, \ldots t_n$. A *subterm* s of t is either t itself, or it is a subterm of a child of t. We write $s = t|_{[i_1, i_2, \ldots i_n]}$ to denote that t has child t_{i_1} which has child t_{i_2} and so on until $t_{i_n} = s$. Note that we can define the recursively as $t|_{[i_1, i_2, \ldots i_n]} = t_{i_1}|_{[i_2, \ldots i_n]}$, which matches our definition for subterm. We call $[i_1, i_2, \ldots i_n]$ the *path* from t to s. We write ϵ for the empty path, and $i : p$ for the path starting with the number i and followed by the path p, and $p \cdot q$ for concatenation of paths p and q.

In our previous term $S(0)$ is a subterm in two different places. One occurrence is at path $[0, 1]$, and the other is at path $[1, 0]$.

We write $t[p \leftarrow r]$ to denote replacing subterm $t|_p$ with r. Algorithmically we can define this as in figure **??**

$t[\epsilon \leftarrow r] = r$
$f(t_1, \ldots t_i, \ldots t_n)[i : p \leftarrow r] = f(t_1, \ldots t_i[p \leftarrow r], \ldots t_n)$

Figure 5: algorithm for finding a subterm of t.

In our above example $t = *(+(0, S(0), +(S(0), 0)))$, We can compute the rewrite $t[[0, 1] \leftarrow *(S(0), S(0))]$, and we get the term $*(+(0, *(S(0), S(0))), +(S(0), 0))$, with the tree in figure **??**.

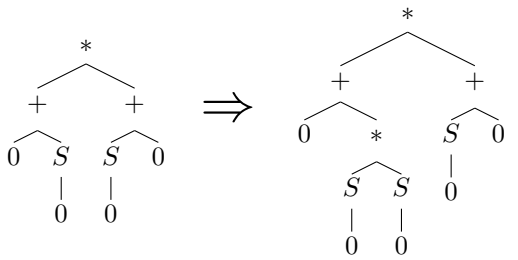


Figure 6: The result of the computation $t[[0, 1] \leftarrow S(0)]$

A substitution replaces variables with terms. Formally a *substitution* is a mapping from $\sigma : V \to T(\Sigma, V)$. We write $\sigma = \{v_1 \mapsto t_1, \ldots v_n \mapsto t_n\}$ to denote the substitution where $s(v_i) = t_i$ for $i \in \{1 \ldots n\}$, and $s(v) = v$ otherwise. We can uniquely extend σ to a function on terms by figure **??**

$\sigma'(v) = \sigma(v)$
$\sigma'(f(t_1, \ldots t_n) = f(\sigma'(t_1) \ldots \sigma'(t_n))$

Figure 7: algorithm for applying a substitution.

Since this extension is unique, we will just write σ instead of σ'. If term t_1 *matches* term t_2 if there exists some substitution σ such that $t_1 = \sigma(t_2)$, Two terms t_1 and t_2 *unify* if there exists some substitution σ such that $\sigma(t_1) = \sigma(t_2)$. In this case σ is called a *unifier* for t_1 and t_2.

We can order substitutions based on what variables they are defined on. a substitution $\sigma \leq \tau$ iff there is some substitution ν such that $\tau = \nu \circ \sigma$. The relation $\sigma \leq \tau$ should be read as σ is more general than τ, and it is a quasi order on the set of substitutions. A unifier u for two terms is *most general* (or an mgu), iff, for all unifiers v, $v \leq u$. Mgus are unique up to renaming of variables. That is, if u_1 and u_2 are mgu's for two terms, then $u_1 = \sigma_1 \circ u_2$. and $u_2 = \sigma_2 \circ u_1$. This can only happen if σ_1 and σ_2 just rename the variables in their terms.

Again as an example $+(x, y)$ matches $+(0, S(0))$ with $\sigma = \{x \mapsto 0, y \mapsto S(0)\}$. The term $+(x, S(0))$ unifies with term $+(0, y)$ with unifier $\sigma = \{x \mapsto 0, y \mapsto S(0)\}$. If $\tau = \{x \mapsto 0, y \mapsto S(z)\}$, then $\tau \leq \sigma$. We can define $\nu = \{z \mapsto 0\}$, and $\{\sigma = \nu \circ \tau\}$

Now that we have a definition for a term, we need to be able to rewrite it. As above a rewrite rule $l \to r$ is a pair of terms. However this time we require that $Var(r) \subseteq Var(l)$, and that $l \notin V$. A Term Rewrite System (TRS) is the pair $(T(\Sigma, V), R)$ where R is a set of rewrite rules.

Definition 1.1. Rewriting Given terms t, s, path p, and rule $l \to r$, we say that t rewrites to s if, l matches $t|_p$ with matcher σ, and $t[p \leftarrow \sigma(r)] = s$. The term $\sigma(l)$ is the *redex*, and the term $\sigma(r)$ is the *contractum* of the rewrite.

There are a few important properties of rewrite rules $l \to r$. A rule is left (right) linear if $l(r)$ is linear. A rule is collapsing if $r \in V$. A rule is duplicating if there is an $x \in V$ that occurs more often in r than in l.

Two terms s and t are *overlapping* if t unifies with a subterm of s, or s unifies with a subterm of t at a non-variable position. Two rules $l_1 \to r_1$ and $l_2 \to r_2$ if l_1 and l_2 overlap. A rewrite system is overlapping if any two rules overlap. Otherwise it's non-overlapping. Any non-overlapping left linear system is *orthogonal*. The following theorem is well known that all orthogonal TRSs are confluent. [**?**]

As an example, in figure **??** examples (b) and (c) both overlap. It's clear that these systems aren't confluent, but non-confluence can arise in more subtle ways. The converse to theorem **??** isn't true. There can be overlapping systems which are confluent.

$g(0, y) \to 0$ $g(1, y) \to 1$	$g(0, y) \to 0$ $g(x, 1) \to 1$	$f(g(x, y)) \to 0$ $g(x, y) \to 1$
(a) A non-overlapping system	(b) A system that overlaps at the root	(c) A system that overlaps at a subterm

Figure 8: Three TRSs demonstrating how rules can overlap. In (a) they don't overlap at all, In (b) both rules overlap at the root, and in (c) rule 2 overlaps with a subterm of rule 1.

When defining rewrite systems we usually follow the constructor discipline. That is, we separate the set $\Sigma = C \uplus F$. C is the set of *constructors*, and F is the set of *function symbols*. Furthermore, for ever rule $l \to r$, the root of l is a function symbol, and every other symbol is a constructor or variable. We call such systems *constructor systems*. As an example, the rewrite system for Peano arithmetic is a constructor system.

The two sets are $C = \{0, S, \top, \bot\}$ and $F = \{+, *, -, \leq\}$, and the root of the left hand side of each rule is a function symbol. In contrast, the SKI system is not a constructor system. While S, K, I can all be constructors, the Ap symbol

$$
\begin{array}{lcll}
R_1 & : & 0 + y & \rightarrow & y \\
R_2 & : & S(x) + y & \rightarrow & S(x + y) \\
R_3 & : & 0 * y & \rightarrow & 0 \\
R_4 & : & S(x) * y & \rightarrow & y + (x * y) \\
R_5 & : & 0 - y & \rightarrow & 0 \\
R_6 & : & S(x) - 0 & \rightarrow & 0 \\
R_7 & : & S(x) - S(y) & \rightarrow & x - y \\
R_8 & : & 0 \leq y & \rightarrow & \top \\
R_9 & : & S(x) \leq 0 & \rightarrow & \bot \\
R_{10} & : & S(x) \leq S(y) & \rightarrow & x < y \\
R_{11} & : & 0 = 0 & \rightarrow & \top \\
R_{12} & : & S(x) = 0 & \rightarrow & \bot \\
R_{13} & : & 0 = S(y) & \rightarrow & \bot \\
R_{14} & : & S(x) = S(y) & \rightarrow & x = y
\end{array}
$$

Figure 9: The rewrite rules for Pe ano Arithmetic with addition, multiplicaton, subtraction, and compari son. All operators use infix notation.

appears in both root and non-root positions of the left hand side of rules. This example will become important for us in Curry. We will do something similar to implement higher order functions. This means that Curry programs won't directly follow the constructor discipline. Therefore, we must be careful when specifying the semantics of function application.

$$
\begin{array}{lcl}
Ap(I, x) & \rightarrow & x \\
Ap(Ap(K, x), y) & \rightarrow & x \\
Ap(Ap(Ap(S, x), y), z) & \rightarrow & Ap(Ap(x, z), Ap(y, z))x
\end{array}
$$

Figure 10: The SKI system from combinatorial logic.

Constructor systems have several nice properties. They are usually easy to analyzed for confluence and termination. For example if the left hand side of two rules don't unify, then they cannot overlap. We don't need to check if subterms overlap. Furthermore any term that consists entirely of constructors and variables is in normal form. For this reason it's not surprising that most functional languages are based on constructor systems.

Finally, we can introduce conditions to rewriting systems. We introduce a new symbol $\top$ to the rewrite systems alphabet's alphabet Σ. A conditional rewrite rule is a rule $l|c \rightarrow r$ where l, c, and r are terms. A term t conditionally rewrites to s with rule $l|c \rightarrow r$ if there exists a path p and substitution σ such that $t_p = \sigma(l)$, $\sigma(c) \rightarrow^* \top$, and $s = \sigma(r)$.

The idea is actually a pretty simple extension. In order to rewrite a term, we must satisfy a condition. If the condition is true, then the rule is applied. In order to simplify the semantics of this system we determine if a condition is true, be rewriting it to the value $\top$. Figure **??** gives an example of a conditional

$$
\begin{array}{lllll}
gcd(x,x) & & \rightarrow & x \\
gcd(x,y) & \mid \quad y \leq x & \rightarrow & gcd(x-y,y) \\
gcd(x,y) & \mid \quad x \leq y & \rightarrow & gcd(x,y-x)
\end{array}
$$

Figure 11: Conditional rewrite system for computing greatest common divisor.

rewrite system for computing greatest common divisor. It uses the rule defined in **??**.

While most treatments of conditional rewriting [**?**, **?**] define a condition as a pair $s = t$ where s and t mutually rewrite to eachother, I chose this definition because it's closer to the definition of Curry, where then condition must reduce to the boolean value `True` or for the rule to apply.

Curry uses conditional rewriting extensively, and efficiently evaluating conditional rewrite systems is the core problem in most functional logic languages. The solution to this problem comes from the theory of narrowing.

1.2 Narrowing

Narrowing was originally developed to solve the problem of semantic unification. The goal was, given a set of equations $E = \{a_1 = b_2, a_2 = b_2, \ldots a_n = b_n\}$ how do you solve the $t_1 = t_2$ for arbitrary terms t_1 and t_2. Here a solution to $t_1 = t_2$ is a substitution σ such that $\sigma(t_1)$ can be transformed into $\sigma(t_2)$ by the equations in E.

As an example let $E = \{*(x+(y,z)) = +(*(x,y),*(x,z))\}$ Then the equation $*(1,+(x,3)) = +(+(*(1,4),*(y,5)),*(z,3))$ is solved by $\sigma = \{x \mapsto +(4,5), y \mapsto 1, z \mapsto 1\}$. The derivation is in figure **??**.

$$
\begin{array}{ll}
\sigma(*(1,+(x,3))) & = \\
*(1,+(+(4,5),3)) & = \\
+(*(1,+(4,5)),*(1,3)) & = \\
+(+(*(1,4),*(1,5)),*(1,3)) & = \\
\sigma(+(+(*(1,4),*(y,5)),*(z,3))) &
\end{array}
$$

Figure 12: Derivation of $*(1,+(x,3)) = +(+(*(1,4),*(y,5)),*(z,3))$ with $\sigma = \{x \mapsto +(4,5), y \mapsto 1, z \mapsto 1\}$.

Unsurprisingly, there is a lot of overlap with rewriting. One of the earlier solutions to this problem was to convert the equations into a confluent, terminating rewrite system. [**?**] Unfortunately this only works for ground terms. That is, terms without variables. However, this idea still has merit. So we want to extend it to terms with variables.

Before when we would rewrite a term t with rule $l \rightarrow r$, we assumed it was a ground term, then we could find a substitution σ that would match a subterm $t|_p$ with l, so that $\sigma(l) = t|_p$. To extend this idea to terms with variables in them, we look for a unifier σ that unifies $t|_p$ with l. This is really the only

change we need to make. However, now we record σ, because it is part of our solution.

Definition 1.2. Narrowing Given terms t, s, path p, and rule $l \to r$, we say that t narrows to s if, l unifies with $t|_p$ with unifier σ, and $t[p \leftarrow \sigma(r)] = s$. We write $t \rightsquigarrow_{p,l\to r,\sigma} s$. We may write $t \rightsquigarrow_\sigma s$ if p and $l \to r$ are clear.

Notice that this is almost identical to the definition of rewriting. The only difference is that σ is a unifier instead of a matcher.

Narrowing gives us a way to solve equations with a rewrite system, but for our purposes it's more important that narrowing allows us to rewrite terms with free variables.

At this point rewrite systems are a nice curiosity, but they are completely impractical. This is because we don't have plan for solving them. In the definition for both rewriting and narrowing we did not specify how to find σ the correct rule to apply, or even what subterm to apply the rule to.

In confluent terminating systems we could simply try every possible rule at every possible position with every possible substitution. Since the system is confluent, we could choose the first rule that could be successfully applied, and since the system is terminating, we'd be sure to finish. This is the best possible case for rewrite systems, and even this is still terribly inefficient. We need a systematic method for deciding what rule should be applied, what subterm to apply it to, and what substitution to use. This is the role of a strategy.

1.3 rewriting strategies

Our goal with a rewriting strategy is to find a normal form for a term. Similarly our goal for narrowing will be to find a normal form and substitution. However, we want to be efficient when rewriting. We would like to use only local information when deciding what rule to select. We would also like to avoid unnecessary rewrites. Consider the term $Ap(Ap(K, I), Ap(Ap(S, Ap(I, I)), Ap(S, Ap(I, I))))$ from the SKI system **??**. It would be pointless to reduce $Ap(Ap(S, Ap(I, I)), Ap(S, Ap(I, I))))$ since $Ap(Ap(K, I, z)$ rewrites to I no matter what z is.

A *Rewriting Strategy* $\mathcal{S} : T(\Sigma, V) \to Pos$ is a function from terms to positions, such that for any term t, $\mathcal{S}(t)$ is a redex. The idea is that $S(t)$ should give us to position to rewrite, and we find the rule that matches $S(t)$.

For orthogonal rewriting systems, there are two common rewriting strategies, innermost and outermost rewriting. Innermost rewriting corresponds to eager evaluation in functional programming. We rewrite the term that matches a rule that is the furthest down the tree. Outermost strategies correspond roughly to lazy evaluation. We rewrite the highest possible term that matches a rewrite rule.

A strategy is normalizing if a term t has a normal form, then the strategy will eventually find it. While outermost rewriting isn't normalizing in general, it is for a large subclass of orthogonal rewrite systems. This matches the intuition from programming languages. Lazy languages can perform computations that would loop forever with an eager language.

While both of these strategies are well understood, we can actually make a stronger guarantee. We want to reduce only the redexes that are necessary to find a normal form. To formalize this we need to understand what can happen as a term is rewritten. Specifically for a redex s that is a subterm of t, how can s change as t is being rewritten. If we're rewriting at position p with rule $l \to r$, then there are 3 cases to consider.
Case 1: we are rewriting s itself. That is, s is the subterm $t|_p$. Then s Disappears entirely.
Case 2: s is either above $t|_p$, or they are completely disjoint. In this case s doesn't change.
Case 3: s is a subterm of $t|_p$. In this case s may be duplicated, or erased, moved, or left unchanged. It depends on the rule is duplicating, erasing, or right linear. These cases can be seen in figure **??** We can formalize this with the notion of descendants with the following definition

Definition 1.3. Descendant Let $s = t|_v$, and $A = l \to_{p,\sigma,R} r$ be a rewrite step in t. The set of descendants of s is given by $Des(s, A)$

$$Des(s, A) = \begin{cases} \emptyset & \text{if } v = u \\ \{s\} & \text{if } p \not\leq v \\ \{t|_{u \cdot w \cdot q} \ : \ r|_w = x\} & \text{if } p = u \cdot v \cdot q \text{ and } t|_v = x \text{ and } x \in V \end{cases}$$

This definition extends to derivation $t \to_{A_1} t_1 \to_{A_2} t_2 \to_{A_2} \ldots \to_{A_n}, t_{n+1}$. $Des(s, A_1, A_2 \ldots A_n) = \bigcup_{s' \in Des(s,A_1)} Des(s', A_2, \ldots A_n)$.

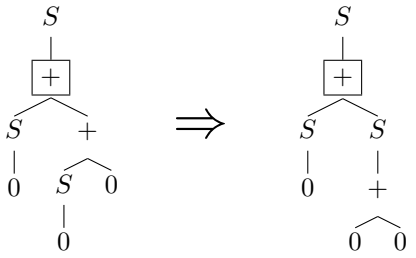
The first part of the definition is formalizing the notion of descendant. The second part is extending it to a rewrite derivation. The extension is straightforward. Calculate the descendants for the first rewrite, then for each descendant, calculate the descendants for the rest of the rewrites. With the idea of a descendant, we can talk about what happens to a term in the future. This is necessary to describing our rewriting strategy. Now, we can formally define what it means for a redex to be necessary for computing a normal form.

Definition 1.4. Needed A redex $s \leq t$ is *needed* if, for every derivation of t to a normal form, a descendant of s is the root of a rewrite.

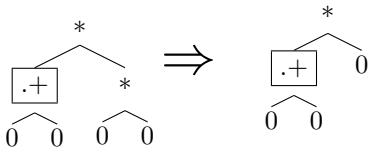
This definition is good because it's immediately clear that is we're going to rewrite a term to normal form, we need to rewrite all of the needed redexes. In fact, we can guarantee more than that with the following theorem.

Theorem 1. *For an orthogonal TRS, any term that is not in normal form contains a needed redex. Furthermore a rewrite strategy that rewrites only needed redexes is normalizing.*

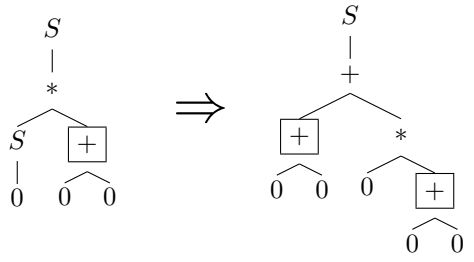
This is a very powerful result. We can compute normal forms by rewriting needed redexes. This is also, in some sense, the best possible strategy. Every needed redex needs to be rewritten. Now we just need to make sure our strategy only rewrites needed redexes. There's only one problem with this plan.



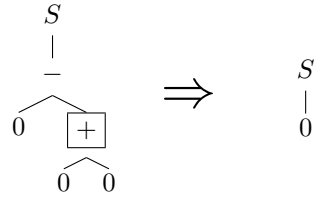
(a) rewrite R_1 at position $[0,1]$ doesn't affect $t|_{[0]}$.



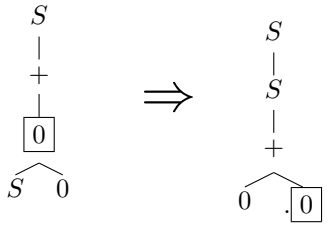
(b) rewrite R_4 at position $[1]$ doesn't affect $t|_{[0]}$.



(c) rewrite R_3 at position $[0]$ duplicates $t|_[0,1]$.



(d) rewrite R_5 at position $[0]$ erases term at $t|_[0,1]$.



(e) rewrite R_2 at position $[0]$ moves $t|_[0,1]$ to position $[0,0,1]$.

Figure 13: four cases for the descendants for a term after a single rewrite. The boxed term is either left alone, duplicated, or erased, or moved.

Determining if a redex is needed is undecidable in general. However, with some restrictions there are rewrite system where this is possible.

Definition 1.5. Sequential A rewrite system is *sequential* if, given a term t with n variables $v_1, v_2 \dots v_n$, such that t is in normal form, then there is an i such that for every substitution σ, $\sigma(v_i)$ is needed in $\sigma(t)$.

If we have a sequential rewrite system, then this leads to an efficient algorithm for reducing terms to normal form. Unfortunately, sequential is also an undecidable property. There is still hope. As we'll see in the next section, with certain restrictions we can ensure the our rewrite systems are sequential. Actually we can make a stronger guarantee. The rewrite system will admit a narrowing strategy that only narrows needed subterms.

1.4 Narrowing Strategies

Similar to rewriting strategies, narrowing strategies attempt to compute a normal form for a term using narrowing steps. However, a narrowing strategy must also compute a substitution for that term. There have been many narrowing strategies including basic [**?**], innermost [**?**], outermost [**?**], standard [**?**], and lazy [**?**]. Unfortunately each of these strategies are too restrictive on the rewrite systems they allow.

$$(x + x) + x = 0$$

(a) This fails for eager narrowing, because evaluating $x + x$ can produce infinitely many answers. However This is fine for lazy narrowing. We will get $(0 + 0) + 0 = 0, \{x = 0\}$ or $S(S(y + S(y)) + S(y)) = 0\{x = S(y)\}$ and the second one will fail.

$$x \leq y + y$$

(b) With a lazy narrowing strategy we may end up computing more than is necessary. If x is instantiated to 0, then we don't need to evaluate $y + y$ at all.

Fortunately there exists a narrowing strategy that's defined on a large class of rewrite systems, only narrows needed expressions, and is sound and complete. However this strategy requires a new construct called a definitional tree.

The idea is straightforward. Since we are working with constructor rewrite systems, we can group all of the rules defined on the same function symbol together. We'll put them together is a tree structure defined below, and when we can compute a narrowing step be traversing the tree for the defined symbol.

Definition 1.6. T is a *partial definitional tree* if T is one of the following.
$T = exempt(\pi)$ where π is a pattern.
$T = leaf(\pi \to r)$ where π is a pattern, and $\pi \to r$ is a rewrite rule.
$T = branch(\pi, o, T_1, \dots T_k)$, where π is a pattern, o is a path, $\pi|_o$ is a variable, $c_1, \dots c_k$ are constructors, and T_i is a pdt with pattern $\pi[c_i(X_1, \dots X_n)]_o$ where n is the arity of c_i, and $X_1, \dots X_n$ are fresh variables.

given a constructor rewrite system R, T is a definitional tree for function symbol f if T is a partial definitional tree, and each leaf in T corresponds to exactly

one rule rooted by f.
A rewrite system is *inductively sequential* if there exists a definitional tree for every function symbol.

The name inductively sequential is justified because there is a narrowing strategy that only reduces needed Redexes for any of these systems. This definition can be difficult to follow mathematically, but it is usually much easier to understand with a few examples. In figure **??** we show the definitional tree for three rules defined above. The idea is that at each branch, we decide which variable to inspect, and we decide what child to follow based on the constructor of that branch. This gives us a simple algorithm for outermost rewriting with definitional trees. However, we need to extend this to narrowing.

The extension from rewriting to narrowing has two complications. The first is that we need to compute a substitution. This is pretty straightforward, but it leads to the second complication. What does it mead for a narrowing step to be needed? The earliest definition involved finding a most general unifier for the substitution. This has some nice properties. There is a well known algorithm for computing mgu's, and they are unique up to renaming of variables. However, this turned out to be the wrong approach. Computing mgu's turns out to be too restrictive. Consider the step $x \leq y + z \rightsquigarrow_{2\cdot\epsilon,R_1,\{y\mapsto 0\}} x \leq z$. Without further substitutions $x \leq z$ is a normal form, and $\{y \mapsto 0\}$ is an mgu. Therefore this should be a needed step. But if we were to instead narrow x, we have $x \leq y + z \rightsquigarrow_{\epsilon,R_8,\{x\mapsto 0\}} \top$. This step never needs to compute a substitution for y. Therefore we need a definition that isn't Dependant on substitutions that might be computed later.

Definition 1.7. A narrowing step $t \rightsquigarrow_{p,R,\sigma} s$ is needed, iff, for every $\eta \geq \sigma$, there is a needed redex at p in $\eta(t)$.

Here we don't require that σ be an mgu, but, for any less general substitution, it must be the case that we're rewriting a needed redex. So out example, $x \leq y + z \rightsquigarrow_{2\dot{\epsilon},R_1,\{y\mapsto 0\}} x \leq z$, isn't a needed narrowing step because $x \leq y + z \rightsquigarrow_{2\dot{\epsilon},R_1,\{x\mapsto 0,y\mapsto 0\}} 0 \leq z$, Isn't a needed rewriting step.

Unfortunately, this definition raises a new problem. Since we are no longer using mgu's for our unifiers, we may not have a unique step for an expression. For example $x < y \rightsquigarrow_{\epsilon,R_8,\{x\mapsto 0\}} \top$, and $x < y \rightsquigarrow_{\epsilon,R_9,\{x\to S(u),t\mapsto S(v)\}} u \leq v$ are both possible needed narrowing steps.

Therefore we define a *Narrowing Strategy* $\mathcal{S}$ as a function from terms to a set of triples of a position, rule, and substitution, such that if $(p, R, \sigma) \in \mathcal{S}(t)$, then $\sigma(t)|_p$ is a redex for rule R.

At this point we have everything we need to define a needed narrowing strategy.

Definition 1.8. Let e be an expression rooted by function symbol f. Let T be

the definitional tree for f.

$$\lambda(e,t) \in \begin{cases} (\epsilon, R, mgu(t,\pi) & \text{if} T = rule(\pi, R) \\ (\epsilon, \bot, mgu(t,\pi) & \text{if} T = exempt(\pi) \\ (p, \bot, \sigma) & \text{if} T = branch(\pi, o, T_1, \ldots T_n) \\ & t \text{unifies with} T_i \\ & (p, R, \sigma) \in \lambda(t, T_i) \\ (p, \bot, \sigma \circ \tau) & \text{if} T = branch(\pi, o, T_1, \ldots T_n) \\ & t \text{does not unifies with any} T_i \\ & T' \text{is the definitional tree for} t|_o \\ & (p, R, \sigma) \in \lambda(t|_o, T') \end{cases}$$

This function lambda simply traverses the definitional tree for a function symbol. If we reach a rule node, then we can just rewrite, if we reach an exempt node, then there is no possible rewrite, if we reach a branch node, then we match a constructor, but if the subterm we're looking at isn't a constructor, then we need to narrow that subterm first.

Theorem 2. *λ is a needed narrowing strategy. Furthermore λ is sound and complete.*

It's should be noted that while lambda is complete with respect to finding substitutions, and selecting rewrite rules, this says nothing about the underlying completeness of the rewrite system we're narrowing. We may still have non-terminating derivations.

This needed narrowing strategy is the underlying mechanism for evaluating Curry programs. In fact, one of the early stages of a curry compiler is to construct definitional trees for each function defined. However, if we were to implement our compiler using terms, it would be needlessly inefficient. we solve this problem with graph rewriting.

1.5 Graph Rewriting

As mentioned above term rewriting is too inefficient to implement curry. In fact all lazy functional language are implemented using graph rewriting. The reason is easy to see. Consider the rule $double(x) = x + x$. This rule would be required to make a copy of x, no matter how large x was. Instead of require that all of our expressions be terms, we relax this restring to include rooted directed graphs. This allows expressions to share variables.

As a brief review of relevant graph theory: A graph $G = (V, E)$ is a pair of vertices V and edges $E \subseteq V \times V$. We will only deal with directed graphs, so the order of the edge matters. A rooted graph is a graph with a specific vertex *root* designated as the root. A path p from vertex u to vertex v is a sequence $u = p_1, p_2 \ldots p_n = v$ where $(p_i, p_{i+1}) \in E$. A rooted graph is connected is there is a path from the root to every other vertex in the graph. A path p is a cycle

[1] if its endpoints are the same. To avoid confusion with variables, we will refer to vertices of graphs as nodes.

We define term graphs in a similar way to terms. Let $\Sigma = C \uplus F$ be an alphabet of constructor and function names respectively, and V be a countably infinite set of variable. A *term graph* is a rooted graph G with nodes in N where each node n has a label in $\Sigma \cup V$. We'll write $L(n)$ to refer to the label of a node. If $(n, s) \in E$ is an edge, then s is a successor of n. In most applications the order of the outgoing edges doesn't matter, however it is very important in term graphs. So, we will refer to the first successor, second successor and so on. The arity of a node is the number of successors. Finally no two nodes can be labeled by the same variable.

While the nodes in a term graph are abstract, in reality, they will be implemented using pointers. It can often be helpful to keep this in mind, as we define more operations on our term graphs, there exists a natural implementation using pointers.

We will often use a linear notation to represent graphs. This has two advantages. The first is that its exactly. Where there are many different ways to draw the same graph. Theres one way to write it out a linear representation. The second is that this representation corresponds closely to the representation in a computer. The notation these graphs is given by the grammar

Graph	$\rightarrow$	Node
Node	$\rightarrow$	n:L(Node, ... Node) $\mid$ n

We start with the root node, and for each node in the graph, If we haven't encountered it yet, then we write down the node, the label, and the list of successors. If we have seen it, then we just write down the node. If a node doesn't have any successors, then we'll omit the parentheses entirely, and just write down the label. A few examples are shown in figure

Let p be a node in G, then the subgraph $G|_p$ is a new graph rooted by p. The nodes are restricted to only those reachable from p. Notice that we node define subgraphs by paths, like we did with subterms. This is because there may be more than one path to the node p. It may be the case that $G|_p$ and G have the same nodes, such as if the root of G is in a loop.

A replacement of node p by graph u in g (written $g[p \leftarrow u]$ is given by the following procedure. For each edge $(n, p) \in E_g$ replace it with an edge $(e, root_u)$. Add all other edges from E_g and E_u. If p is the root of g, then $root_u$ is now the root.

A homomorphism h from g_1 to g_2 is a function where if $n \in g_1$, then $h_n \in g_2$, furthermore if $L(n) \in \Sigma$, then $L(n) = L(h(n))$, and if $(n, s) \in E_{g_1}$, then $(h(n), h(s)) \in E_{g_2}$, and $h(root_{g_1}) = h(root_{g_2})$. That is, the function preserves all non-variable nodes, successors, and the root. A homomorphism is variable preserving if, for all nodes n labeled by a variable in g_1, $L(n) = L(h(n))$.

While the labels will remain the same, the structure of the underlying graph may change significantly under a homorophism, as shown in figure.

[1]Some authors will use walk and tour and reserve path and cycle for the cases where there are no repeated vertices. This distinction isn't relevant for our work.

An Isomorphism